# Akka & Akka Streams

Construa sistemas distribuídos com atores

Casa do Código | alura

ALEXANDRE LOURENÇO

# Sumário


- ISBN
- Agradecimentos
- Sobre o autor
- 1. Era uma vez um sistema...
- 2. Paralelismo: o que é e por que você deveria se interessar
- 3. Trabalhando com atores no Akka
- 4. Ciclo de vida e tratativa de erros em atores
- 5. Construindo uma API: parte 1 - criando a persistência e serviços
- 6. Construindo uma API: parte 2 - criando os endpoints e testando
- 7. Introdução ao Akka Streams
- 8. Integrações do mundo real com Akka Streams
- 9. Testes automatizados com Akka e Akka Streams
- 10. Conclusão

# ISBN

Impresso: 978-65-86110-90-6

Digital: 978-65-86110-91-3


Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

# Agradecimentos

Agradeço a Deus pela oportunidade de estar vivo e atuante. Desejo do fundo do coração toda felicidade, saúde e sucesso a todos que estiverem lendo este livro.

Agradeço a meus pais, Maria Odete e Eleutério. Eu não seria nada sem vocês. Muito obrigado por me suportarem e ensinarem tudo o que me fez ser o que eu sou! Obrigado também a minha irmã, Lucebiane, te amo maninha! Obrigado por me aturar todos esses anos, hehe.

Agradeço a minha amada esposa, Ana Carolina. Sou muito feliz de ter te encontrado em minha vida, que muitos anos mais venham pela frente.

Agradeço à editora Casa do Código e a todos os que me ajudaram na produção deste livro, em especial a Iago David dos Santos, que fez a revisão técnica deste livro. Sua contribuição foi muito importante para garantir a maior qualidade possível das informações apresentadas.

E obrigado a você, leitora ou leitor, que comprou este livro. Espero que eu possa compartilhar com você o melhor que eu puder de meus conhecimentos neste campo. Conhecimento é um dos bens mais preciosos que existem, infelizmente tão

marginalizado nos dias de hoje. Citando uma frase de que gosto muito, do antigo filósofo Sócrates:

"*Existe apenas um bem, conhecimento, e apenas um mal, ignorância*".

## Sobre o livro

Este livro se destina a desenvolvedores e desenvolvedoras que buscam expandir seus conhecimentos em arquiteturas distribuídas. No mundo em que vivemos, onde cada vez mais temos quantidades insanas de dados trafegando para todos os lados, precisamos entender cada vez mais conceitos que nos permitam processar esses dados com agilidade.

Isso é especialmente verdade em integrações, nas quais temos que constantemente sincronizar dados de um sistema A para um sistema B. Tais sincronismos sempre serão parte da vida de quem desenvolve, seja para integrar o sistema do RH com o controle de acessos de funcionários, ou o módulo de vendas de e-commerce com o ERP Fiscal.

Neste livro, experimentaremos arquiteturas distribuídas com casos que simulam sistemas reais, como integrações com sistemas de arquivos, comunicação com Kafka e bancos de dados, entre outros. Construiremos também uma API REST

fictícia de um e-commerce, vendo um exemplo completo de uma aplicação Akka na prática.

Para essa finalidade, aprenderemos sobre os frameworks Akka e Akka Streams e de que forma eles podem nos ajudar a cumprir essa missão.

Akka utiliza o modelo de atores, um modelo que nos permite construirmos aplicações robustas e escaláveis, através de processamento assíncrono e threads. Aprenderemos sobre paralelismo, entendendo de que modo ele nos ajuda a melhorar a escalabilidade de nossos sistemas.

Akka Streams é um framework construído em cima do Akka, que nos permite desenvolver soluções usando o paradigma de streams. Esse paradigma é muito popular em sistemas distribuídos, sendo bastante utilizado junto de ferramentas como o Apache Kafka.

Ele nos permite pensar em soluções como fluxos de dados, permitindo que as soluções possam executar de modo bastante escalável. Para esse fim, Akka Streams é uma ótima alternativa por permitir que implementemos esse paradigma em cima de um framework já consolidado como o Akka.

## Para quem é o livro

Para melhor aproveitar este livro, é recomendado que você tenha conhecimentos básicos de linguagem de programação, bem como o uso de Docker.

Conhecimentos em Scala também podem ajudar, dado que será a linguagem utilizada, entretanto, apenas a sintaxe básica será necessária, e todos os conceitos que venham a ser utilizados serão devidamente explicados, fazendo analogias com a linguagem Java. Portanto, não é necessário ser um expert na linguagem para acompanhar!

Se quiser saber mais sobre Scala, a Casa do Código tem um livro sobre essa linguagem: *Scala: Como escalar sua produtividade*, de Paulo Siqueira.

# Sobre o autor

Formado em Ciência da Computação pela PUC-SP com pós-graduação pelo UNIBTA, Alexandre Eleutério Santos Lourenço trabalha com desenvolvimento de software desde 2003. Atuou com diversos frameworks, ferramentas e linguagens, como Elasticsearch, Kafka, Docker, Kubernetes, Java, Python, Scala, entre outras e, é claro, Akka Streams, tendo trabalhado com a tecnologia em desenvolvimentos ligados a integrações e sistemas de geolocalização.

## Capítulo 1
# Era uma vez um sistema...

Era uma vez um pacato desenvolvedor que vivia em uma empresa de e-commerce em construção. Ele cuidava do sistema de estoque, sua vida era tranquila e feliz - tirando eventuais *outages* aqui e ali -, pois, até aquele momento, o seu querido sistema vivia pacato como uma ilha: o desenvolvimento do módulo de estoque tinha sido concluído e já estava em seus primeiros tempos de uso, mas não havia outros sistemas maduros o suficiente para se comunicar com ele.

Um belo dia, a gerente do pacato desenvolvedor o chamou até a sala. Ela lhe disse que o sistema de vendas finalmente ficou pronto! Exultante, o desenvolvedor logo pensou nas novas possibilidades e experiências com que ele teria contato.

A equipe do sistema de vendas solicitou ao desenvolvedor que fizesse uma API REST para permitir que eles consultassem quantas unidades havia no estoque para um determinado produto e para obter um item do estoque (remover uma unidade de um produto do estoque).

Alegremente, o desenvolvedor construiu uma interface para que o time de vendas pudesse utilizar o estoque. Ele construiu uma API Java, com o framework Spring Boot. Após concluir o desenvolvimento, ele testou os endpoints. Para consultar a quantidade de unidades disponíveis no estoque para um dado produto, ele forneceu o seguinte endpoint:

```
GET - /stock/{id}
10
```

Que, no nosso exemplo, indica que o produto possui dez unidades disponíveis. Para reservar um produto, alocando para o comprador uma unidade de um produto, ele construiu o seguinte endpoint:

```
POST - /stock/{id}
true
```

Que, no nosso exemplo, retornou o valor `true` representando o sucesso da operação. Ele pensou:

"*Minha única preocupação como estoque é a concorrência, e já tenho o meu banco relacional para me proteger disso. Essa foi fácil! HAHAHA!*".

Por que a preocupação com concorrência? Se temos, por exemplo, dois clientes tentando obter o mesmo produto do estoque, temos o perigo de gerar uma inconsistência se não garantirmos que ambas

as atualizações sejam feitas com o valor atual do estoque.

Nesse tipo de soluções é comum utilizarmos bancos de dados relacionais, que através de mecanismos de locks transacionais garantem que sempre realizemos atualizações consistentes (sempre atualizando o valor mais atual do estoque, neste exemplo).

Deu-se início ao uso da API. Durante o período normal de vendas do ano, o sistema de estoque aguentou bem - havia uma ou outra reclamação da parte do time de vendas de que os endpoints eram lentos, mas alguns índices de banco e *query hints* aliviaram a situação - então a paz relativamente reinava no nosso pequeno ambiente.

Chegou a Black Friday. Todos estavam apreensivos, afinal, é um dos períodos mais massivos de vendas no ano, devido aos grandes descontos nos produtos. Começaram as vendas. Após algum tempo, o sistema de vendas começou a demonstrar lentidão. Ao investigar, descobriu-se a causa: a quantidade massiva de chamadas ao estoque estava causando a lentidão.

Sob tensão, nosso intrépido desenvolvedor começou sua investigação. Detectando que o banco era a causa do gargalo, foi acionado o time de infraestrutura, que adicionou mais memória e CPU

ao banco. Isso remediou um pouco a situação, mas só um pouco.

Após algum tempo, o desenvolvedor começou a notar algo estranho: ainda que as monitorações do banco estivessem altas, elas se mantinham estáveis, porém os tempos de resposta do sistema aumentavam mais e mais, indo para vários segundos por requisição.

Por infelicidade, o time do sistema de vendas também se esqueceu de configurar devidamente um timeout para as suas chamadas. Ou seja, quando uma requisição ocorre, o sistema de vendas fica "preso", esperando para sempre enquanto não obtém uma resposta da requisição. Assim sendo, quanto mais chamadas ele fazia ao estoque, mais preso o sistema ficava aguardando as respostas das chamadas.

A lentidão no sistema de estoque ocorria principalmente devido à arquitetura tradicional de servidores web. Nessa arquitetura tradicional, um *pool* de *threads* (aprenderemos o que é isso em breve) atende às requisições dos clientes do servidor, neste caso, o sistema de vendas.

Como todas as threads do servidor estavam ocupadas esperando pela resposta do banco, quando a quantidade massiva de chamadas

externas chegava, não havia threads que pudessem atender a essas chamadas. Nessa situação, a tendência desse tipo de servidor é começar a demorar mais e mais para responder às requisições.

Conforme o quadro vai se agravando, ele pode começar a rejeitar as requisições (com HTTP errors, como o famoso *504 - Gateway Timeout*) até que, se a onda de chamadas não se aplacar, o sobrecarregado servidor vem abaixo.

Foi exatamente isso o que aconteceu: o sistema de estoque não resistiu e caiu com a carga de requisições. Como o sistema de vendas se encontrava na mesma situação, por cascata, ele também não resistiu e acabou indo abaixo na sequência.

Depois de vários *restarts* e uma noite atribulada, finalmente a quantidade de requisições aplacou e os sistemas foram voltando ao normal. Nossos heróis da tecnologia foram para suas casas com tristeza em seus corações, pensando nos clientes que não conseguiram fazer suas compras e no que fazer para que da próxima vez os sistemas estejam mais bem preparados.

O que aprendemos com essa história? Desenvolver software é uma tarefa complexa, que exige pensar em vários aspectos do sistema. Ao focar toda a sua

preocupação no controle transacional - o que é uma preocupação muito importante - nosso herói se esqueceu de pensar em outros aspectos, como avaliar (se possível, até mesmo testar com alguma ferramenta de *load testing* ou *stress testing*) a performance do sistema em si.

Não existe sistema com escalabilidade infinita. A instância de uma aplicação possui um limite finito que ela pode suportar, a partir do qual outras técnicas como escalabilidade horizontal podem nos ajudar.

Um dos maiores inimigos da performance de uma aplicação são as operações de I/O. Na nossa história, as operações de I/O eram feitas pelo banco de dados. Bancos de dados relacionais são ótimas soluções, porém, são notórios pela sua performance mais degradada com relação a outras partes de um sistema. Para aliviar o problema, é necessário isolar essas operações, de modo a deixá-las impactar a menor parte possível da infraestrutura. Talvez se a envolvermos com algum tipo de camada especial...

Para aprendermos a melhorar a performance de nossas aplicações, construiremos, no capítulo 5, a API desse e-commerce, mas, antes disso, precisamos entender alguns conceitos. Convido você agora a me acompanhar nessa jornada, entrando no mundo do paralelismo.

## Capítulo 2

# Paralelismo: o que é e por que você deveria se interessar

Para melhorar a performance e robustez de nossas aplicações, uma das técnicas mais poderosas que temos à nossa disposição é o paralelismo computacional. Vamos aprender os seus conceitos, navegando através de um exemplo prático, no qual observaremos como ele funciona na prática.

Neste capítulo, assim como no restante do livro, usaremos a linguagem Scala. Baseada na JVM, essa robusta linguagem permite o desenvolvimento tanto utilizando o paradigma de Programação Funcional quanto o de Orientação a Objetos, além de ser a linguagem-padrão para se programar em Akka e Akka Streams. Utilizaremos a versão 2.13.6 da linguagem.

> Também é possível programar em Java, mas recomendo muito que você experimente a linguagem Scala através dos exemplos que mostraremos neste livro, garanto que você gostará!

Para execução/compilação/testes utilizaremos a ferramenta de build sbt. Feita pela LightBend (mantenedora também do Akka), ela se tornou a ferramenta de build padrão da linguagem. É possível instalar o sbt seguindo as instruções em https://www.scala-sbt.org/1.x/docs/Setup.html.

Todo o código-fonte do projeto se encontra no repositório Git do livro. Nele, temos um projeto sbt já configurado, prontinho para ser utilizado. O endereço é: https://github.com/alexandreesl/livro-akka-streams.

Complementando o ambiente de desenvolvimento, como IDE, sugiro utilizar o Intellij Community Edition. O código utilizará a versão 2.13.6 da linguagem, rodando sob a JVM Zulu 16.1.0.

Vamos começar aprendendo sobre paralelismo definindo o que são processos e threads.

## 2.1 Processos

Para entender o que são processos, vamos criar um programa em Scala que consiste em um simples print no terminal. Vamos criar no nosso projeto um arquivo chamado `application.scala` e codificar o seguinte:

```
object Application {

  def main(args: Array[String]): Unit = {
    println("Hello word!")

    while(true) {

    }

  }

}
```

Vamos executar a aplicação rodando o seguinte comando com o sbt:

```
sbt run
```

Isso vai produzir o seguinte resultado no console, além de manter o terminal em suspensão devido ao loop infinito que colocamos no final do programa.

```
copying runtime jar...
[info] welcome to sbt 1.5.5 (Azul Systems, Inc. Java
16.0.1)
[info] loading settings for project global-plugins from
idea.sbt ...
[info] loading global plugins from
/Users/alexandrelourenco/.sbt/1.0/plugins
[info] loading project definition from
/Users/alexandrelourenco/livro-akka-streams/project
[info] loading settings for project livro-akka-streams
from build.sbt ...
[info] set current project to akka-streams (in build
```

```
file:/Users/alexandrelourenco/livro-akka-streams/)
[info] compiling 1 Scala source to
/Users/alexandrelourenco/livro-akka-
streams/target/scala-2.13/classes ...
[info] Non-compiled module 'compiler-bridge_2.13' for
Scala 2.13.6. Compiling...
[info]   Compilation completed in 8.176s.
[info] running Application
Hello word!
```

Vamos abrir um novo terminal, mantendo o que está executando o nosso programa, e executar um `grep` no sistema operacional, a fim de observarmos os programas em execução.

*Os comandos de sistema operacional deste livro assumem um ambiente Unix/Linux.*

```
ps aux | grep sbt
```

Uma grande linha de texto no terminal será produzida, começando com algo parecido com isto:

```
alexandrelourenco 26112 100.0  4.4  7293136 733628 s002
R+    8:55PM  16:38.51 java -Dfile.encoding=UTF-8 -
Xms1024m ........
```

Nessa linha, temos diversas informações referentes ao programa em execução, como o seu ID de sistema operacional, quantidade de CPU e memória sendo consumida, data e hora de início da execução do programa etc.

Dentro do sistema operacional, programas em execução são chamados de **processos**. Processos consomem recursos, utilizam drivers etc. De todas as unidades de processamento dentro de uma máquina, processos são os mais custosos, que mais demandam esforço computacional para serem instanciados. Por conta disso, devem ser utilizados com sabedoria, utilizando bem os recursos que alocam do sistema operacional.

Processos são a principal forma de paralelismo no nível do sistema operacional. Cada processo é executado de forma independente e simultânea pelo sistema operacional, com a CPU se desdobrando para garantir que o máximo possível de processos estejam executando ao mesmo tempo.

Ok! Então entendemos o que são processos. Mas o que são threads?

## 2.2 Threads

Quando um processo é iniciado, o sistema operacional nos fornece o recurso de criação de threads. Threads são como subprocessos dentro de um mesmo processo. Diferentemente de processos, threads compartilham a CPU e memória alocados pelo processo.

Por conta do compartilhamento de recursos, threads compartilham informações entre si, ou seja, podemos ter informações em memória sendo compartilhadas entre as threads (o que deve ser tratado com cuidado para evitar problemas, como veremos em breve). Threads são mais leves do que processos, pois são criadas e removidas com menos custos.

Vamos começar a observar como uma thread se comporta na prática. Vamos modificar o objeto que fizemos anteriormente da seguinte forma:

> Em Scala, temos o conceito de classes e objetos. Objetos são parecidos com classes, porém, diferentemente de classes, objetos são instâncias únicas dentro do interpretador Scala, que não podem ser reinstanciadas. O mais próximo que podemos interpretar analogamente um objeto Scala para o universo Java seria uma classe instanciada sob o padrão Singleton.

```
import java.lang.Thread.{currentThread, sleep}
import java.time.LocalDateTime

object Application {

  def main(args: Array[String]): Unit = {

    val minhaThread: Runnable = () => {
```

```
      while (true) {
        println(s"executando ${LocalDateTime.now} na
thread ${currentThread().getName}")
        sleep(1000L)
      }
    }

    val thread = new Thread(minhaThread)
    thread.run()

  }

}
```

Após a execução, podemos ver uma série de mensagens demonstrando que a thread foi criada e está sendo executada:

```
[info] compiling 1 Scala source to
/Users/alexandrelourenco/livro-akka-
streams/capitulo_2/target/scala-2.13/classes ...
[info] running Application
executando 2021-07-16T20:28:36.682992 na thread run-
main-0
executando 2021-07-16T20:28:37.690419 na thread run-
main-0
executando 2021-07-16T20:28:38.692532 na thread run-
main-0
executando 2021-07-16T20:28:39.697353 na thread run-
main-0
executando 2021-07-16T20:28:40.699886 na thread run-
main-0
```

Mas será mesmo que threads são tão mais rápidas assim do que uma pura e simples execução serial

(single-thread)? Vamos fazer um teste!

Vamos começar modificando o nosso programa para fazer uma simples execução, sem threads. Nessa execução, simularemos um processamento mais custoso, que leva alguns segundos para executar - o processamento de uma grande pilha de números - e logar quanto tempo levará para que a execução seja realizada.

Vamos mudar o código da seguinte forma:

```
import java.lang.Thread.currentThread
import java.time.LocalDateTime
import scala.collection.mutable

object Application {

  def main(args: Array[String]): Unit = {

    println(s"executando (inicio do processamento)
${LocalDateTime.now}")

    @volatile var stack = mutable.Stack[Int]()
    stack.pushAll(List.range(1,9990000))

    while (stack.nonEmpty) {
      stack.pop()
    }

    println(s"executando (termino do processamento)
${LocalDateTime.now}")

  }
```

```
}
```

No código anterior, utilizamos a anotação `@volatile`. Essa anotação define que as operações de escrita/leitura do nosso objeto - no caso, a colocação/remoção de números na pilha - sejam feitas de forma atômica, ou seja, o Scala garante para nós um bloqueio no acesso aos métodos dessa variável, de modo que apenas uma thread por vez possa acessá-la. Essa anotação é um recurso análogo ao *synchronized*, que veremos mais à frente no capítulo.

Após a execução com o `sbt run`, teremos algo parecido com isto no console (obviamente, seu resultado dependerá de fatores como a potência do seu computador):

```
[info] welcome to sbt 1.5.5 (Azul Systems, Inc. Java
16.0.1)
[info] loading settings for project global-plugins from
idea.sbt ...
[info] loading global plugins from
/Users/alexandrelourenco/.sbt/1.0/plugins
[info] loading project definition from
/Users/alexandrelourenco/livro-akka-
streams/capitulo_2/project
[info] loading settings for project capitulo_2 from
build.sbt ...
[info] set current project to akka-streams (in build
```

```
file:/Users/alexandrelourenco/livro-akka-
streams/capitulo_2/)
[info] compiling 1 Scala source to
/Users/alexandrelourenco/livro-akka-
streams/capitulo_2/target/scala-2.13/classes ...
[info] running Application
executando (inicio do processamento) 2021-07-
19T19:29:03.398409
executando (termino do processamento) 2021-07-
19T19:29:06.536477
[success] Total time: 6 s, completed Jul 19, 2021,
7:29:06 PM
```

Como é possível observar, o processamento leva alguns segundos para executar. Vamos modificar o código acrescentando duas threads, que vão se revezar no processamento da pilha:

```
import java.lang.Thread.currentThread
import java.time.LocalDateTime
import java.util.concurrent.Executors
import scala.collection.mutable
import scala.concurrent.duration.MINUTES

object Application {

  def main(args: Array[String]): Unit = {

    println(s"executando (inicio do processamento)
${LocalDateTime.now}")

    @volatile var stack = mutable.Stack[Int]()
    stack.pushAll(List.range(1, 9990000))

    val implementacao: Runnable = () => {
      if (stack.nonEmpty) stack.pop()
```

```
    }

    val servicoExecucao = Executors.newCachedThreadPool

    servicoExecucao.execute(implementacao)
    servicoExecucao.execute(implementacao)

    servicoExecucao.shutdown()

    servicoExecucao.awaitTermination(1, MINUTES)

    println(s"executando (termino do processamento)
${LocalDateTime.now}")

  }

}
```

Executando o código, vemos um código como este:

```
[info] welcome to sbt 1.5.5 (Azul Systems, Inc. Java
16.0.1)
[info] loading settings for project global-plugins from
idea.sbt ...
[info] loading global plugins from
/Users/alexandrelourenco/.sbt/1.0/plugins
[info] loading project definition from
/Users/alexandrelourenco/livro-akka-
streams/capitulo_2/project
[info] loading settings for project capitulo_2 from
build.sbt ...
[info] set current project to akka-streams (in build
file:/Users/alexandrelourenco/livro-akka-
streams/capitulo_2/)
[info] compiling 1 Scala source to
/Users/alexandrelourenco/livro-akka-
streams/capitulo_2/target/scala-2.13/classes ...
```

```
[info] running Application
executando (inicio do processamento) 2021-07-
19T21:12:34.853257
executando (termino do processamento) 2021-07-
19T21:12:37.810829
[success] Total time: 6 s, completed Jul 19, 2021,
9:12:37 PM
```

Ué, o que aconteceu? Levou o mesmo tempo para executar do que com apenas uma thread!

Isso é um ponto importante que devemos ter em mente quando trabalhamos com paralelismo: se pensarmos puramente em tempo de execução, *multithread* só começa a fazer alguma diferença a partir de volumes mais massivos de dados. Em quantidades menores, podemos observar basicamente o mesmo tempo de execução que um processamento de uma única thread.

Evidentemente, outras questões devem ser consideradas quando utilizamos paralelismo. Em um exemplo real, o processamento dos dados em si não consiste apenas em remover números de uma pilha, mas outras operações, como efetuar chamadas a outras APIs.

Se temos chamadas a uma API que leva 1s para retornar, evidentemente um processamento em threads será mais rápido, dado que fará chamadas

simultâneas à API, em vez de fazer uma por vez, como no processamento single thread.

Entendemos que paralelismo consiste na execução simultânea de blocos de código. Ao executarmos código em paralelo, permitimos uma melhor performance, já que podemos processar dados em simultâneo em vez de processarmos um dado de cada vez. Quantos blocos de código podem ser executados simultaneamente no máximo? Infinitos? Infelizmente, essa não é a resposta.

Dentro da arquitetura moderna de CPUs, cada unidade de processamento (ou core de CPU) pode executar um código por vez. Assim, nosso limite de execução em paralelo é equivalente à quantidade de cores de nossa CPU.

O diagrama a seguir demonstra de maneira macro como funciona a estrutura de processos e threads dentro de um sistema operacional:

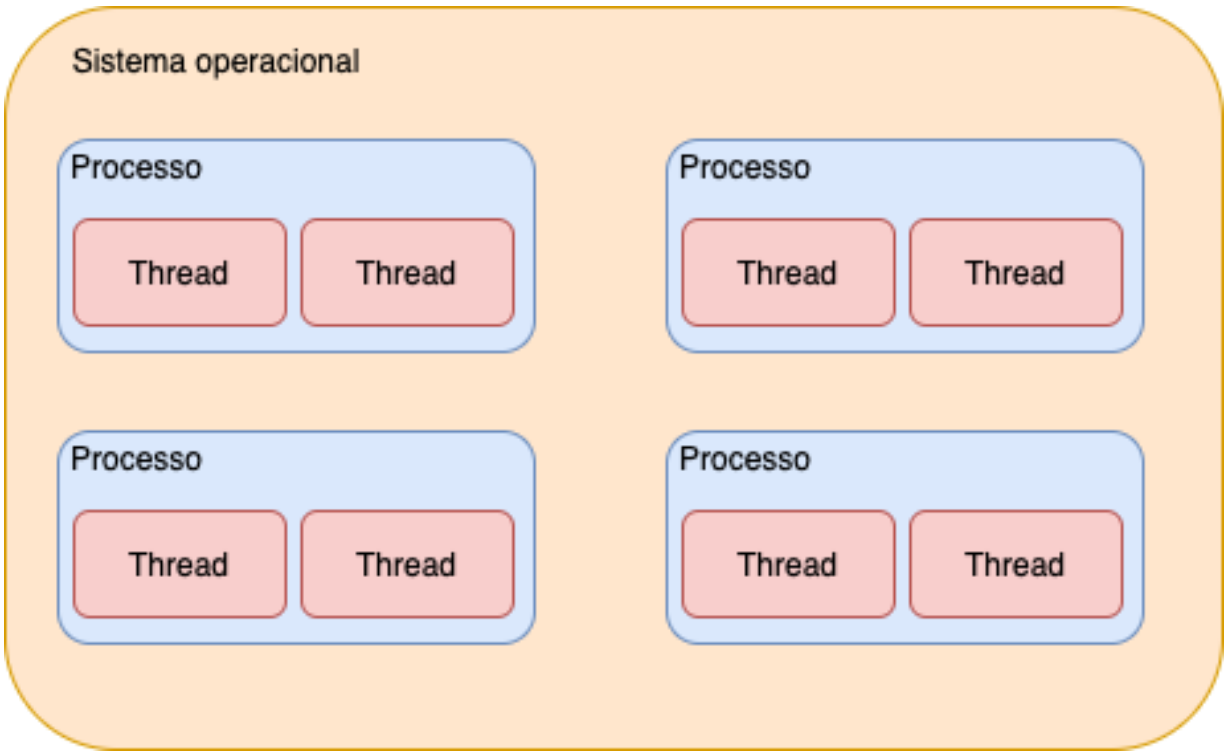



Figura 2.1: Estrutura de processos e threads dentro do sistema operacional.

## 2.3 Concorrência e paralelismo

No meu ambiente, possuo uma CPU com quatro cores. O que isso significa? Vamos supor que eu tenha um programa que instancie três threads. A figura a seguir representa as execuções das threads nos cores da CPU:

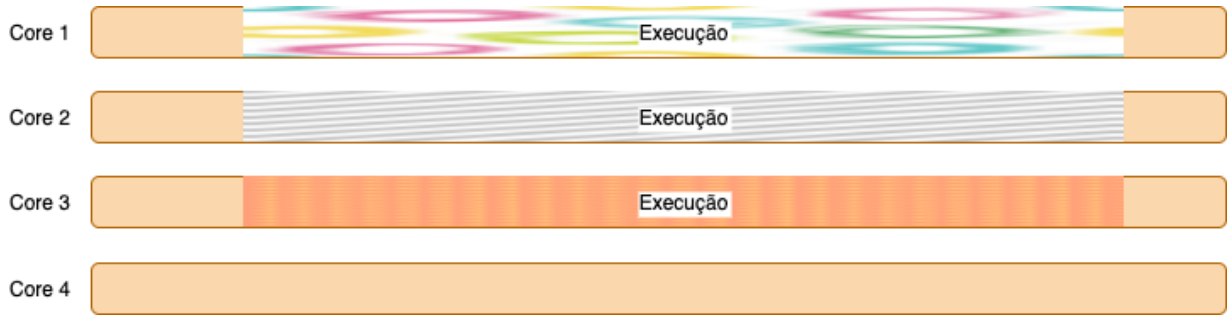



Figura 2.2: Execução em uma quantidade de threads menor que a quantidade de cores.

Agora, o que acontece se temos uma quantidade de threads maior do que a quantidade de cores disponível? A figura a seguir exibe um diagrama de como seria tal cenário, no qual temos cinco threads em execução para os mesmos quatro cores:

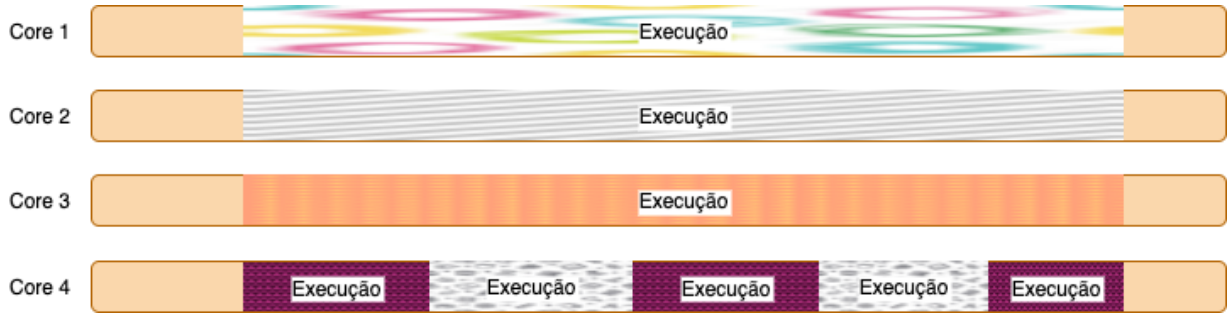



Figura 2.3: Execução em uma quantidade de threads maior que a quantidade de cores.

O que aconteceu? Nesse cenário, temos as threads de número 4 e 5 se alternando na execução, com 1 core executando as duas threads de forma

alternada. A esse processamento alternado damos o nome de **concorrência**.

E assim temos a diferença entre concorrência e paralelismo. Na concorrência, temos diversas threads sendo executadas por um número limitado de cores e as execuções alternadas ocorrem de forma muito rápida, porém não de forma simultânea. No paralelismo, temos de fato código sendo executado simultaneamente através da arquitetura física de processamento da CPU.

Anteriormente, comentamos sobre como threads permitem que informações na memória sejam compartilhadas entre si. Vamos ver na prática esse compartilhamento e seus perigos.

## 2.4 Compartilhamento de dados entre threads

Vamos fazer um outro exemplo simples de uso de threads. Suponhamos que temos duas threads atualizando um contador, uma thread efetuando subtrações e a outra, adições. Cada operação será executada 120 vezes, sem qualquer controle ou ordenação das operações:

```
import java.time.LocalDateTime
import java.util.concurrent.Executors
import scala.collection.mutable
```

```
import scala.concurrent.duration.MINUTES

object Application {

  def main(args: Array[String]): Unit = {

    var contador: Int = 0

    val somador = () => {
      contador += 1
      contador
    }

    val redutor = () => {
      contador -= 1
      contador
    }

    val operador = (operacao: () => Int,
identifcadorOperacao: String) => {
      for (i <- List.range(1, 120))
println(f"executando a operacao" +
        f" $identifcadorOperacao - valor do contador
${operacao()}")
    }

    val servicoExecucao = Executors.newCachedThreadPool

    servicoExecucao.execute(() => {
      operador(somador, "soma")
    })
    servicoExecucao.execute(() => {
      operador(redutor, "subtração")
    })

    servicoExecucao.shutdown()
```

```
    servicoExecucao.awaitTermination(1, MINUTES)

  }
```

Nesse código, podemos ver conceitos de Programação Funcional aplicados na prática. A fim de reutilizar código, temos uma função que recebe por parâmetro outra função com o intuito de envelopar a função dentro do loop de execuções, junto com o print das informações. Tais funções são chamadas de funções de primeira ordem.

Após executar o código, podemos observar este fragmento:

```
...
executando a operacao soma - valor do contador 2
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao subtração - valor do contador 1
executando a operacao soma - valor do contador 2
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador -1
executando a operacao soma - valor do contador 0
...
```

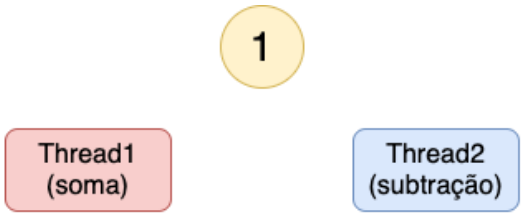
Podemos observar um comportamento estranho: a atualização do contador é totalmente caótica, com operações aparentemente "atualizando o mesmo valor", além de "saltos" aqui e ali no valor do contador, como no fragmento a seguir:

```
...
executando a operacao subtração - valor do contador -9
executando a operacao subtração - valor do contador -10
executando a operacao soma - valor do contador -4
executando a operacao subtração - valor do contador -11
executando a operacao soma - valor do contador -10
executando a operacao subtração - valor do contador -11
executando a operacao soma - valor do contador -10
executando a operacao subtração - valor do contador -11
executando a operacao soma - valor do contador -10
executando a operacao subtração - valor do contador -11
...
```

Por que isso acontece? Para entender isso, precisamos relembrar do que falamos no começo do capítulo, sobre compartilhamento de recursos entre threads.

Como dito anteriormente, threads compartilham recursos. Assim, quando temos duas threads acessando a mesma informação, temos uma disputa pelo recurso. Nesse processo de disputa, acessos e escritas simultâneos ocasionam a geração de resultados imprevisíveis. A figura a seguir demonstra, em uma linha do tempo, um exemplo desse tipo de problema:

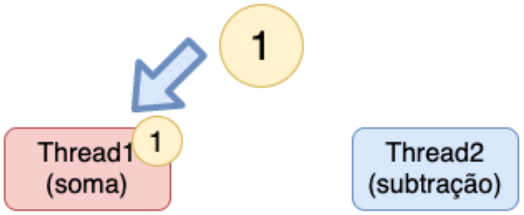
Momento 1:




As duas threads estão prestes a obter o valor da variável (1)

---

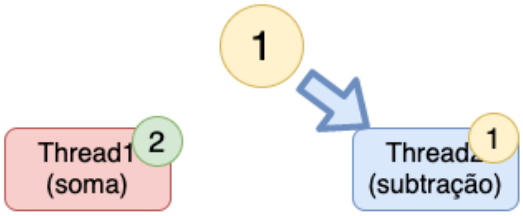
Momento 2:




A thread de soma obtém o valor da variável, gerando uma cópia local da variável para efetuar o trabalho

---

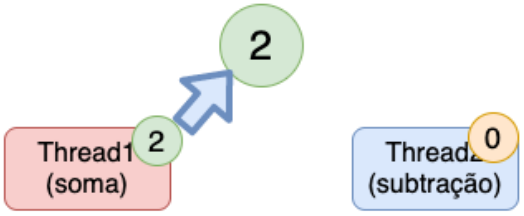
Momento 3:




A thread de soma atualiza o seu valor local. Enquanto isso, a thread de subtração obtém a sua cópia local da variável

---

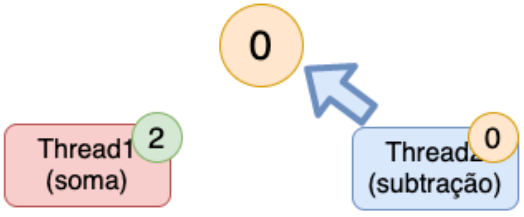
Momento 4:




A thread de soma atualiza a variável. Enquanto isso, a thread de subtração atualiza a sua versão - antiga - da variável

---

Momento 5:




A thread de subtração atualiza, de forma incorreta, o valor da variável utilizando a sua versão da informação

Figura 2.4: Atualização de uma variável compartilhada por duas threads em uma linha do tempo.

Como resolvemos esse problema? Em linguagens de programação modernas, possuímos alguns recursos que permitem "bloquear" o acesso de threads às informações compartilhadas, por exemplo, o famoso *synchronized* do Java (também presente no Scala). Com esses recursos, podemos garantir que apenas uma thread por vez acesse o recurso, garantindo que apresente o comportamento esperado.

Ainda que neste simples exemplo resolvamos o problema com uma técnica de bloqueio, esta não é considerada a melhor forma de se resolver problemas de concorrência. Realizar todo o controle de acesso das threads aos recursos demanda esforço computacional por parte do ambiente de execução, além de minar o propósito geral de se ter uma solução distribuída. Uma solução considerada mais moderna para a persistência de informações nesse tipo de ambiente são os chamados **OBJETOS IMUTÁVEIS**. Quando falarmos mais sobre o modelo de atores no decorrer do livro, aprenderemos mais sobre eles.

Outro recurso que pode ser utilizado são os chamados **objetos atômicos** (*atomic objects*). Objetos atômicos permitem que operações feitas com eles ocorram de modo atômico, ou seja, que apenas uma única thread possa realizar suas operações em um determinado instante.

Vamos alterar o exemplo anterior, utilizando agora um objeto atômico:

```
import java.time.LocalDateTime
import java.util.concurrent.Executors
import java.util.concurrent.atomic.AtomicInteger
import scala.collection.mutable
import scala.concurrent.duration.MINUTES

object Application {

  def main(args: Array[String]): Unit = {

    var contador: AtomicInteger = new AtomicInteger(0)

    val somador = () => {
      contador.addAndGet(1)
    }

    val redutor = () => {
      contador.addAndGet(-1)
    }

    val operador = (operacao: () => Int,
identifcadorOperacao: String) => {
      for (i <- List.range(1, 120))
println(f"executando a operacao" +
        f" $identifcadorOperacao - valor do contador
```

```
${operacao()}")
    }

    val servicoExecucao = Executors.newCachedThreadPool

    servicoExecucao.execute(() => {
      operador(somador, "soma")
    })
    servicoExecucao.execute(() => {
      operador(redutor, "subtração")
    })

    servicoExecucao.shutdown()

    servicoExecucao.awaitTermination(1, MINUTES)

  }
```

Executando novamente, podemos observar uma execução dentro do esperado:

```
...
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
executando a operacao subtração - valor do contador 0
executando a operacao soma - valor do contador 1
...
```

## 2.5 Deadlocks

Outro problema bastante famoso no mundo das threads é o famigerado **deadlock**. Deadlocks ocorrem quando threads acessando recursos compartilhados entram em um estado de bloqueio, pois uma thread está esperando a liberação de um recurso que outra thread está bloqueando, que por sua vez está esperando por outro recurso que está sendo bloqueado por outra thread (ou pela própria thread aguardando a liberação). Vamos ver um exemplo simples:

```
import java.time.LocalDateTime
import java.util.concurrent.Executors
import java.util.concurrent.atomic.AtomicInteger
import scala.collection.mutable
import scala.concurrent.duration.MINUTES

object Application {

  class Recurso(val name: String) {
    def preparar(recurso: Recurso): Unit = {
      synchronized {
        println(s"sou ${name} e estou me preparando
para utilizar ${recurso.name}")
        recurso.utilizando(this)
      }
    }

    def utilizando(recurso: Recurso): Unit = {
      synchronized {
        println(s"sou ${name} e estou utilizando
${recurso.name}")
```

```
    }
  }
}

def main(args: Array[String]): Unit = {

  val recursoA = new Recurso("Recurso A")
  val recursoB = new Recurso("Recurso B")
  new Thread(new Runnable() {
    override def run(): Unit = {
      recursoA.preparar(recursoB)
    }
  }).start()
  new Thread(new Runnable() {
    override def run(): Unit = {
      recursoB.preparar(recursoA)
    }
  }).start()

}
```

Ao executar, vamos observar que a aplicação não executou até o fim, entrando em um estado de lock infinito, que só se encerra se forçarmos o programa a terminar com *Ctrl*+*C*:

```
[info] running Application
sou Recurso A e estou me preparando para utilizar
Recurso B
sou Recurso B e estou me preparando para utilizar
Recurso A
```

O deadlock ocorreu porque o recurso A tentou acessar o recurso B ao mesmo tempo em que o recurso B tentou acessar o recurso A. Isso gerou

uma trava entre as threads, onde uma está aguardando a liberação de acesso da outra. Quando falarmos sobre o modelo de atores, vamos entender como seguir suas diretrizes ajuda a nos protegermos desse problema.

E assim concluímos a nossa pequena jornada pelo mundo do paralelismo. Agora que temos os conceitos devidamente solidificados, vamos começar a conhecer o modelo de atores!

## Capítulo 3
# Trabalhando com atores no Akka

Agora que entendemos o que é paralelismo, vamos começar a trabalhar com atores. O modelo de atores usa e abusa do paralelismo, sendo uma excelente opção na construção de APIs, como veremos futuramente na nossa API de e-commerce.

## 3.1 Atores

O modelo de atores foi idealizado por Carl Hewit, Peter Bishop e Richard Steiger em 1973. O paper deles pode ser encontrado no link https://www.ijcai.org/Proceedings/73/Papers/027B.pdf.

Nesse modelo, atores são unidades de programação dentro de um programa, instanciadas dentro de um *servidor de atores*. Esses atores se comunicam entre si através de *mensagens*, que funcionam como no padrão DTO (*Data Transfer Object*). Essas mensagens são trafegadas através de *canais*, que são como buffers de mensagens sendo enfileiradas para leitura pelos atores.

Um ponto importante sobre as mensagens é que elas devem ser criadas como objetos *imutáveis*, assim não temos problemas com locks e acessos compartilhados de informações como vimos anteriormente, pois os objetos não podem ser modificados, sendo sempre criadas novas cópias.

Toda essa comunicação é feita de forma assíncrona, ou seja, quando uma mensagem é enviada para um ator, ela é depositada para a posterior leitura do ator em seu canal (também chamada de *mailbox* (caixa de correio)). Isso também é verdade na comunicação entre atores: em um cenário em que um ator A necessita se comunicar com um ator B (e receber um retorno de B), esse retorno deve ser feito através de outra mensagem, que deve ser enviada a partir do ator B para o ator A.

Esses recursos de processamento assíncrono, com buffers de mensagens entre os atores, nos proveem uma série de vantagens:

- Temos um modelo em que cada operação do processamento é encapsulada em um ator diferente. Desse modo, conseguimos codificar, testar, refatorar etc. cada parte de forma totalmente independente, sem afetar as outras operações;
- Essa linha de raciocínio também nos protege dos famigerados deadlocks, como vimos

anteriormente. Cada ator é desenvolvido sob o conceito *stateless*, em que seu escopo cabe apenas o processamento das mensagens que lhe são recebidas, utilizando objetos imutáveis para encapsular os dados. Desse modo, não temos deadlocks, pois não temos compartilhamento de acesso às mesmas informações, ou informações inconsistentes por estarem sendo acessadas de modo concorrente;
- Com os buffers, podemos modelar o fluxo de execução de modo que as operações mais lentas possam ser executadas em paralelo com as operações mais rápidas, desse modo, ganhamos muito no tempo de execução dos programas;
- Também é possibilitado, com o mecanismo de buffers, que o processamento mais custoso, como operações de acesso a recursos (como a bancos de dados, como vimos no exemplo do capítulo 1), seja isolado em atores específicos, o que desse modo não impacta todo o processamento.

Ainda sobre deadlocks, você pode estar se perguntando se não continuaremos tendo dados compartilhados quando desenvolvermos um sistema que acessa um banco de dados, por exemplo. Nesse caso específico, é verdade que, na camada de banco de dados, haverá um mecanismo de controle a fim de garantir a consistência das informações. Porém, ao eliminarmos esse controle da camada de aplicação, empurrando-o o máximo possível "para fora" da aplicação (no caso, apenas no banco de dados), aliado com o isolamento do acesso em um ator, como citamos, isso faz com que esse controle seja minimizado e isolado o máximo possível, e seu impacto torna-se bastante baixo.

Todo esse controle é feito pelo sistema de atores, que encapsula a criação de pools de threads, onde os atores são executados. Esses pools de threads criados pelo sistema de atores consistem em grupos de instâncias de um componente chamado *Dispatcher*. Dispatchers são os executores da lógica implementada pelos atores, responsáveis por obter mensagens dos canais e executar a sua lógica, além de implementar toda lógica relacionada ao consumo das mensagens, como os mecanismos de retentativas em caso de falhas - veremos mais sobre

isso no próximo capítulo, no qual veremos como o Akka implementa tratativas de erros em atores.

O diagrama a seguir ilustra o relacionamento entre o sistema de atores, canais, dispatchers e os atores em si:

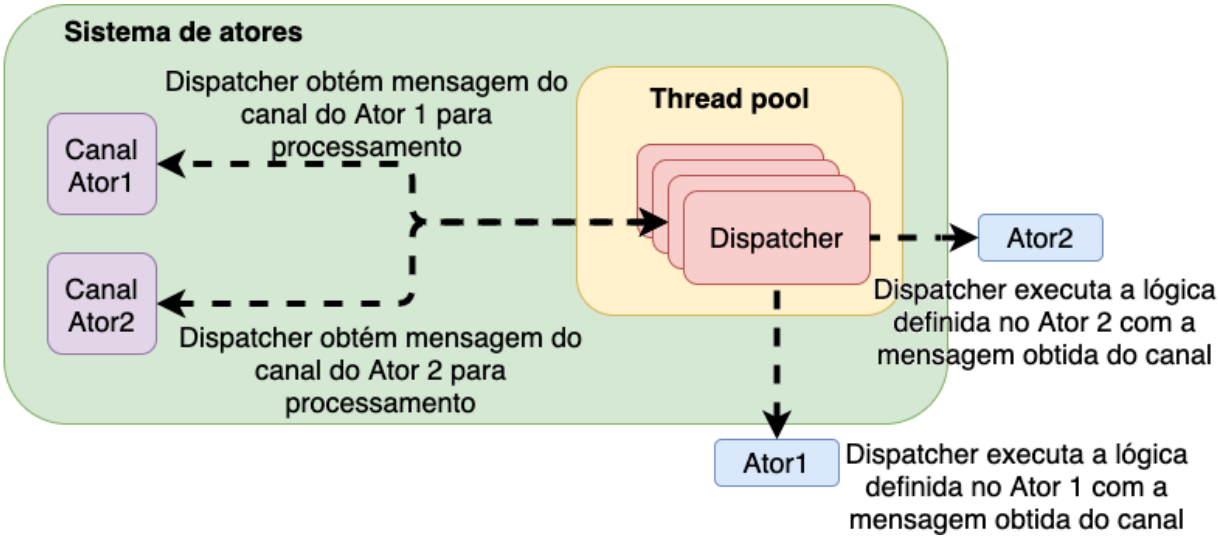

Sistema de atores
Dispatcher obtém mensagem do canal do Ator 1 para processamento
Thread pool
Canal Ator1
Canal Ator2
Dispatcher
Ator2
Dispatcher executa a lógica definida no Ator 2 com a mensagem obtida do canal
Dispatcher obtém mensagem do canal do Ator 2 para processamento
Ator1
Dispatcher executa a lógica definida no Ator 1 com a mensagem obtida do canal

Figura 3.1: Sistema de atores com seus dispatchers, canais e atores.

No decorrer do livro, abordaremos outros conceitos, como o ciclo de vida de atores e os supervisores. Por ora, vamos trabalhar os conceitos básicos, começando com exemplos simples.

## 3.2 Nosso primeiro ator

Nosso primeiro exemplo simulará um processamento simples, onde temos dois atores, um simulando um acesso a uma API e outro simulando a persistência dos dados em uma base, após a chamada a API.

Para começar, temos que incluir as dependências do Akka no projeto. Vamos modificar o arquivo `build.sbt` , conforme a seguir:

```
name := "akka-streams"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test
)

// dependencias de logging
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)
```

Vamos começar criando dois atores simples. No nosso modelo, um ator receberá uma mensagem e, em seguida, enviará a mensagem para o segundo ator, representando uma sequência de

processamento de uma mensagem por diferentes atores. O exemplo simbolizará um exemplo real, em que cada ator desempenhará uma função diferente: um ator chamará uma API externa, e o seguinte salvará os dados em um banco de dados.

Vamos começar com o ator que grava a mensagem no banco de dados:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors

object DBAtor {

  case class MensagemBanco(nome: String, documento:
String)

  def apply(): Behavior[MensagemBanco] =
Behaviors.receive {
    (contexto, mensagem) =>
      contexto.log.info(s"mensagem $mensagem recebida e
persistida no banco!")
      Behaviors.same
  }

}
```

No código, podemos ver uma `case class`. Case classes são um recurso poderoso da linguagem Scala, que permite criarmos classes estilo POJO (*Plain Old Java Object*, ou os bons e velhos objetos Java, como getters/setters, toString, entre outros) com pouquíssimo código. Somente com a inclusão da palavra-chave *case*, já temos getters, setters, toString, hashCode e equals devidamente implementados!

Criar um ator é bem simples. No modelo funcional, podemos utilizar objetos para a criação dos atores, dentro do método `apply` (o método `apply` é invocado na inicialização do objeto).

O método de fábrica `Behaviors.receive` chamado no código consiste em uma fábrica de atores, na qual construímos uma função que recebe como parâmetros o contexto - objeto utilitário que possui diversos recursos, como um logger sistêmico já configurado para logar no console - e a mensagem em si. É possível ter múltiplos tipos de mensagens sendo processadas por um mesmo ator, como veremos em breve.

Vamos criar agora outro objeto que realize o setup do sistema de atores para já fornecer uma forma de executar o ator:

```
package com.casadocodigo.actors

import akka.NotUsed
import akka.actor.typed.{Behavior, Terminated}
import akka.actor.typed.scaladsl.Behaviors

object ActorSetup {

  def apply(): Behavior[NotUsed] = Behaviors.setup {
    _ =>
      Behaviors.receiveSignal {
        case (_, Terminated(_)) =>
          Behaviors.stopped
      }
  }
}
```

Neste setup, utilizar o método de fábrica `Behaviors.setup` nos permite realizar configurações no sistema, como criar atores, permitindo criar atores filhos (vamos entender isso melhor em breve). No nosso exemplo, configuramos, com o `Behaviors.receiveSignal` , que o sistema de atores será encerrado corretamente caso a aplicação seja encerrada.

A seguir, vamos recodificar a função `main` do nosso projeto. Vamos instanciar o servidor de atores e enviar uma mensagem para o nosso ator:

```
import akka.actor.typed.ActorSystem
import com.casadocodigo.actors.{ActorSetup, DBActor}
import com.casadocodigo.actors.DBActor.MensagemBanco
```

```
object Application {

  def main(args: Array[String]): Unit = {

    val system = ActorSystem(ActorSetup(),
"MyActorSystem")

    val atorDB =
system.systemActorOf(DBActor(),"DBActor")

    atorDB ! MensagemBanco(nome = "teste", documento =
"123456")

  }

}
```

Na função, vemos como é simples instanciar o servidor. Basta criar um objeto, passando um objeto chamado *ActorSetup*, contendo as configurações padrão de um servidor de atores, e uma string, que é o nome do servidor. Assim, é possível criar múltiplos servidores em uma mesma aplicação se necessário, por exemplo, em casos em que precisamos de variações de um mesmo ator executando para diferentes fluxos de processamento.

Depois, criamos o ator dentro do sistema de atores. Isso é feito através do método `systemActorOf`, passando a implementação e uma string identificadora do ator no sistema.

Por fim, realizamos o envio da mensagem. Esse envio é feito através do operador `!`. O operador `!` é uma **sobrecarga** do método `tell`, ou seja, na prática, se enviássemos a mensagem da forma abaixo, o resultado seria o mesmo:

```
atorDB.tell(MensagemBanco(nome = "teste", documento = "123456"))
```

Em Scala, todos os operadores da linguagem são funções e/ou métodos de classes. Assim, é possível criar funções e métodos com nomes que permitem criar *syntax sugars* para simplificar o código, como no exemplo acima em que `!` é uma simplificação para chamadas ao método `tell`.

A linguagem ainda possui o recurso chamado **implícitos**, onde podemos sobrescrever operadores já definidos dentro do escopo de nosso código, até mesmo os operadores padrões da linguagem! Tal recurso, porém, deve ser usado com certa cautela, a fim de não prejudicar a compreensão do código para desenvolvedores sem contexto do código, ou mesmo a perda do controle da aplicação (imagine colocar um + em uma operação aritmética e não saber o que vai acontecer!).

Ao executar o código, teremos a seguinte mensagem no terminal, indicando que nosso sistema de atores subiu com sucesso:

```
20:01:13.129 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO akka.event.slf4j.Slf4jLogger -
Slf4jLogger started
SLF4J: A number (1) of logging calls during the
initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the
filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay
20:01:13.164 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO com.casadocodigo.actors.DBActor$ -
mensagem MensagemBanco(teste,123456) recebida e
persistida no banco!
```

Agora que criamos nosso primeiro ator, é importante notar que ele simboliza um fluxo de processamento. Ou seja, não faria sentido enviarmos uma mensagem para esse ator, sem passar primeiro pelo ator da chamada a APIs. Como podemos montar uma estrutura que reflita essa dependência? Para isso, vamos compreender como funciona uma hierarquia de atores para então criarmos nosso segundo ator, aquele que simboliza a chamada à API externa.

## Hierarquia de atores

Como citado anteriormente, no exemplo que estamos construindo, temos uma relação entre os

atores, na qual um deve receber apenas mensagens pré-processadas pelo outro. Além disso, não faz sentido que um ator esteja instanciado sem que o outro também esteja. Como resolver essa questão?

Dentro de um servidor de atores, temos a hierarquia de atores. Atores criados diretamente no servidor (os chamados atores de sistemas, ou *System actors*) estão no primeiro nível da hierarquia, enquanto atores criados dentro de outros atores são chamados de **atores filhos**.

> Dentro do sistema de atores, existe um ator implícito criado pelo sistema, chamado de ator raiz. Assim, os atores de sistema também são atores filhos.

Por que criar atores filhos? Além da questão já citada de encapsulamento do código, atores filhos são associados aos seus pais, que também são chamados de ***supervisores**. Como veremos futuramente, supervisores são muito importantes, permitindo que desenvolvamos tratativas de erro específicas por grupos de atores. Outra função desses agrupamentos é permitir um melhor controle, assim podemos, por exemplo, reiniciar os atores quando necessário (entenderemos isso quando falarmos sobre o ciclo de vida deles).

## 3.3 Nosso segundo ator

Vamos criar o nosso segundo ator, refatorando o código para que ele seja o supervisor do nosso primeiro ator:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors
import com.casadocodigo.actors.DBActor.MensagemBanco

object APIActor {

  case class MensagemAPI(nome: String, documento:
String)

  def apply(): Behavior[MensagemAPI] = Behaviors.setup
{ contexto =>

    val refAtorDB = contexto.spawn(DBActor(),
"DBActor")

    Behaviors.receive {
      (contexto, mensagem) =>
        contexto.log.info(s"mensagem $mensagem recebida
e enviada para a API!")
        refAtorDB ! MensagemBanco(nome = mensagem.nome,
documento = mensagem.documento)
        Behaviors.same
    }
  }

}
```

E modificar a função `main` para utilizar o novo ator:

```
import akka.actor.typed.ActorSystem
import com.casadocodigo.actors.APIActor.MensagemAPI
import com.casadocodigo.actors.{APIActor, ActorSetup}

object Application {

  def main(args: Array[String]): Unit = {

    val system = ActorSystem(ActorSetup(),
"MyActorSystem")

    val atorAPI = system.systemActorOf(APIActor(),
"APIActor")

    atorAPI ! MensagemAPI(nome = "teste", documento =
"123456")

  }

}
```

Após executarmos, podemos ver no log que os atores executaram o fluxo com sucesso:

```
[info] running Application
19:04:59.459 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO akka.event.slf4j.Slf4jLogger -
Slf4jLogger started
SLF4J: A number (1) of logging calls during the
initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the
filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay
19:04:59.523 [MyActorSystem-akka.actor.default-
```

```
dispatcher-7] INFO com.casadocodigo.actors.APIActor$ -
mensagem MensagemAPI(teste,123456) recebida e enviada
para a API!
19:04:59.524 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.DBActor$ -
mensagem MensagemBanco(teste,123456) recebida e
persistida no banco!
```

Sucesso! Temos o nosso primeiro exemplo funcional, com dois atores se comunicando. Algumas considerações finais sobre o nosso primeiro exemplo:

- Conforme dito anteriormente, toda a comunicação entre os atores é assíncrona. Quando chamamos o envio da mensagem para o ator, a aplicação não é bloqueada. Uma forma simples de comprovar esse aspecto é invertendo as linhas que enviam e logam a mensagem, assim podemos ver nos horários que as duas instruções ocorreram em simultâneo:

```
19:14:35.945 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO com.casadocodigo.actors.APIActor$ -
mensagem MensagemAPI(teste,123456) recebida e enviada
para a API!
19:14:35.945 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO com.casadocodigo.actors.DBActor$ -
mensagem MensagemBanco(teste,123456) recebida e
persistida no banco!
```

- Cada ator é executado por uma thread diferente. Dentro dos meandros do sistema de atores,

temos diferentes *thread pools* executando tarefas, como o processamento do código dos atores em si, limpeza de buffers de mensagens etc. Podemos verificar que cada ator executa seu código dentro da sua thread por meio dos logs, onde temos informações, como *MyActorSystem-akka.actor.default-dispatcher-5*.

Antes de seguirmos para outro exemplo, vamos observar como seria o mesmo exemplo utilizando Orientação a Objetos, em vez de Programação Funcional. Não existe nenhuma diferença prática na execução, apenas uma outra forma de se desenvolver com atores.

Vamos começar desenvolvendo a versão OO do nosso ator que "persiste" no banco de dados:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.{AbstractBehavior,
ActorContext}
import com.casadocodigo.actors.DBActor.MensagemBanco

class DBClassActor(contexto:
ActorContext[MensagemBanco]) extends
AbstractBehavior(contexto) {
  override def onMessage(msg: MensagemBanco):
Behavior[MensagemBanco] = msg match {
    case MensagemBanco(nome, documento) =>
      contexto.log.info(s"mensagem ${nome} -
${documento} recebida e persistida no banco!")
```

```
      this
  }
}
```

Alguns pontos a se observar no nosso novo formato:

- No exemplo, estamos estendendo a classe `AbstractBehavior`, que fornece o suporte a classes para serem transformadas em atores.
- A seguir, estendemos o método `onMessage`, que receberá as mensagens enviadas para o ator. O restante do método em si consiste no mesmo processamento de antes: logamos a mensagem simbolizando a persistência da mensagem no banco de dados.

Em Scala, temos uma forma um pouco diferente de utilizar construtores. Quando colocamos código entre ( ) logo após o nome de uma classe, estamos definindo os parâmetros do seu construtor padrão.

E ao definir parâmetros após o nome da classe que estamos estendendo, estamos definindo que o nosso construtor deve invocar o construtor da classe pai repassando os parâmetros recebidos. Prático, não?

Outro recurso do Scala que estamos vendo pela primeira vez é o **pattern matching**. Esse poderoso recurso - recém-adicionado ao Java na versão 16 - permite inserirmos trechos de código em cláusulas, conforme a tipagem da mensagem, por exemplo (veremos como isso é imensamente útil quando tivermos exemplos com múltiplos tipos de mensagens sendo processadas pelo mesmo ator).

Podemos ver também que conseguimos acessar os atributos da mensagem diretamente, sendo fornecidos através de variáveis criadas na própria cláusula. Esse recurso é conhecido como desempacotamento (*unboxing*) de objetos.

A seguir, vamos desenvolver a versão OO do nosso ator que "envia" os dados para a API:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.{AbstractBehavior,
ActorContext, Behaviors}
import com.casadocodigo.actors.APIActor.MensagemAPI
import com.casadocodigo.actors.DBActor.MensagemBanco

class APIClassActor(contexto:
ActorContext[MensagemAPI]) extends
AbstractBehavior(contexto) {

  val refAtorDB = contexto.spawn(Behaviors.setup {
    context: ActorContext[MensagemBanco] =>
      new DBClassActor(context)
  }, "DBClassActor")

  override def onMessage(msg: MensagemAPI):
Behavior[MensagemAPI] = msg match {
    case MensagemAPI(nome, documento) =>
      contexto.log.info(s"mensagem ${nome} -
${documento} recebida e enviada para a API!")
      refAtorDB ! MensagemBanco(nome = nome, documento
= documento)
      this
  }
}
```

Como podemos observar, não há grandes diferenças com relação ao já explicado na nossa refatoração anterior, tirando que agora, para realizarmos a criação do ator filho, utilizamos o

`Behaviors.setup` como uma função anônima para instanciar o ator com o contexto tipado corretamente.

Finalmente, vamos modificar a função `main`, instanciando também o nosso ator estilo OO e enviando uma mensagem diferente para ele:

```
import akka.actor.typed.ActorSystem
import akka.actor.typed.scaladsl.{ActorContext,
Behaviors}
import com.casadocodigo.actors.APIActor.MensagemAPI
import com.casadocodigo.actors.{APIActor,
APIClassActor, ActorSetup}

object Application {

  def main(args: Array[String]): Unit = {

    val system = ActorSystem(ActorSetup(),
"MyActorSystem")

    val atorAPI = system.systemActorOf(APIActor(),
"APIActor")

    atorAPI ! MensagemAPI(nome = "teste", documento =
"123456")

    val atorClassAPI =
system.systemActorOf(Behaviors.setup {
      contexto: ActorContext[MensagemAPI] =>
        new APIClassActor(contexto)
    }, "APIClassActor")

    atorClassAPI ! MensagemAPI(nome = "teste 2",
documento = "123456789")
```

```
  }

}
```

Após rodar o código, podemos ver que ambas as mensagens foram processadas corretamente:

```
20:31:13.837 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.APIClassActor - mensagem teste
2 - 123456789 recebida e enviada para a API!
20:31:13.837 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO com.casadocodigo.actors.APIActor$ -
mensagem MensagemAPI(teste,123456) recebida e enviada
para a API!
20:31:13.837 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO com.casadocodigo.actors.DBClassActor
- mensagem teste 2 - 123456789 recebida e persistida no
banco!
20:31:13.838 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO com.casadocodigo.actors.DBActor$ -
mensagem MensagemBanco(teste,123456) recebida e
persistida no banco!
```

Agora que temos o nosso primeiro exemplo funcional (com o perdão do trocadilho) devidamente compreendido, vamos experimentar algo mais complexo.

## 3.4 Múltiplas mensagens e observadores (watchers)

Vamos seguir agora para o próximo exemplo, que explorará recursos mais avançados do Akka, como atores processando múltiplas mensagens e observadores *(watchers)*.

Vamos imaginar um modelo de atores que represente um módulo de controle remoto de um robô de exploração espacial (se você não sabe o que é um robô de exploração espacial, segue uma imagem, de um que foi enviado pela NASA para Marte).

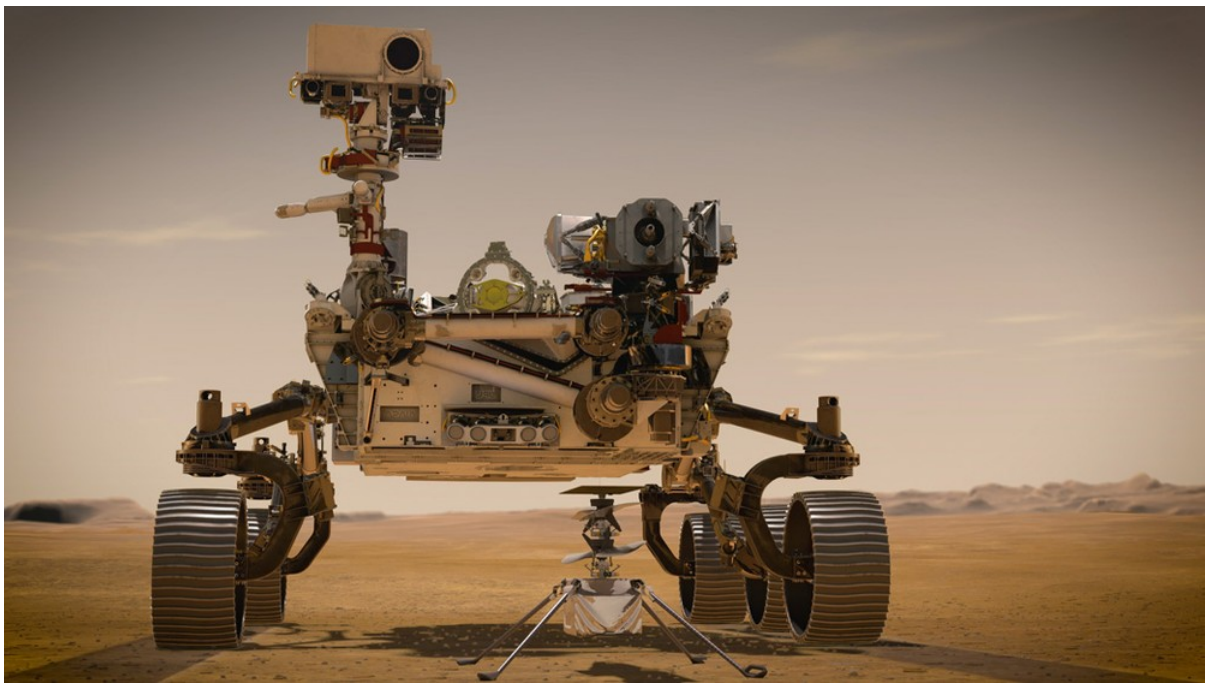


Figura 3.2: Robô Perseverance da NASA.

No nosso exemplo, temos diferentes possibilidades de comandos que podemos emitir para o robô:

- **Ir para frente**: como o próprio nome diz, esse comando faz o robô ir para a frente. Por padrão, o robo anda 1m para a frente;
- **Ir para trás**: esse comando faz o robô ir para trás;
- **Ir para a esquerda**: rotaciona o robô 90 graus para a esquerda e anda 1m;
- **Ir para a direita**: rotaciona o robô 90 graus para a direita e anda 1m;
- **Coletar amostra**: faz o robô utilizar seus equipamentos de coleta para obter amostras do solo. Nosso robô possui um limite de quatro amostras de solo sem realizar a transmissão dos resultados - se uma nova coleta for solicitada, o coletor de amostras deve ser finalizado e as amostras transmitidas antes que ele possa ser inicializado novamente, do contrário, os circuitos de coleta do robô vão fritar;
- **Transmitir análises**: faz o robô enviar os resultados das suas análises em cima do solo coletado.

Com os nossos comandos devidamente definidos, vamos começar o exemplo!

## Criando a hierarquia de comandos

Vamos começar criando a hierarquia de classes que representam os comandos. Essa hierarquia encapsulará os dados e estruturará os comandos que permitirão a configuração dos nossos atores:

```
package com.casadocodigo.actors.robot

import java.util.UUID


sealed trait ComandoDeMovimento

case class MoverParaFrente extends ComandoDeMovimento
case class MoverParaTras extends ComandoDeMovimento
case class MoverParaEsquerda extends ComandoDeMovimento
case class MoverParaDireita extends ComandoDeMovimento

sealed case class Coleta(UUID: UUID, peso: Double)

case class Coletar(coleta: Coleta)
case class IniciarTransmissao(coleta: Coleta)
case class Transmitir(coleta: Coleta)
```

No código acima, podemos ver dois outros conceitos da linguagem Scala: *traits* e *sealed*. Traits são parecidas com interfaces do Java, porém elas permitem definir comportamento (métodos/funções com implementações), diferentemente das interfaces clássicas (pré-Java 8). Traits podem ser extensões de outras traits, e classes podem estender múltiplas traits (mas apenas uma classe). Sealed é um modificador de acesso da linguagem, que define que traits e classes definidas como sealed só podem ser estendidas por outras traits e classes que sejam definidas dentro do mesmo arquivo Scala.

## Desenvolvendo um ator que processa múltiplas mensagens

Após definirmos os comandos, vamos começar a desenvolver os atores. Vamos começar com o ator responsável por processar os comandos de movimentos:

```
package com.casadocodigo.actors.robot

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors

object MoveRobotActor {

  def apply(): Behavior[ComandoDeMovimento] =
```

```
Behaviors.receive {
    (contexto, mensagem) =>
      mensagem match {
        case MoverParaFrente =>
          contexto.log.info(s"Robô movido para a frente
com sucesso!")
          Behaviors.same
        case MoverParaTras =>
          contexto.log.info(s"Robô movido para trás com
sucesso!")
          Behaviors.same
        case MoverParaEsquerda =>
          contexto.log.info(s"Robô movido para a
esquerda com sucesso!")
          Behaviors.same
        case MoverParaDireita =>
          contexto.log.info(s"Robô movido para a
direita com sucesso!")
          Behaviors.same
      }
  }

}
```

Como podemos ver, é um ator bastante simples. Uma novidade que podemos notar é que se trata do nosso primeiro ator que processa múltiplas mensagens. Tal recurso é muito simples de se implementar: basta desenvolvermos uma hierarquia de mensagens com traits/classes base para os comandos similares, e configurarmos atores que operem sob as classes e traits base, utilizando-se de *pattern matching* para implementar os

processamentos de acordo com a mensagem enviada.

## Processando mensagens em lote com StashBuffer

Vamos agora desenvolver o ator que realiza as coletas. Relembrando o nosso escopo, temos um limite de quatro coletas simultâneas no máximo. Caso recebamos mais do que isso, devemos transmitir para a base de operações tudo o que se encontra já coletado e encerrar o coletor de amostras, a fim de evitar que os sistemas do robô fritem.

> Para emular o buffer de coletas, utilizaremos um recurso do Akka chamado *StashBuffer*. Esse componente consiste de uma pilha *in-memory* não thread-safe, que deve ser utilizada apenas para casos simples de dados perenes, em que não temos problemas com perdas de informações.

Um ponto importante a se destacar sobre o thread-safety, que citamos na nota acima, é que isso **não** é um problema no nosso caso, pois vamos processar o buffer através do recebimento de mensagens do Akka.

Conforme já aprendemos, o processamento dos atores é executado em thread pools, de forma assíncrona. Porém, o framework nos garante que, em um dado momento, sempre teremos **apenas uma thread executando cada ator do sistema de atores**. Assim, não há problema em se utilizar esse tipo de estrutura sem proteção para acessos de diferentes threads, desde que a utilização seja confinada dentro do ambiente de execução do ator.

A imagem a seguir exemplifica o ambiente de execução do Akka no esquema threads X atores, conforme explicado:

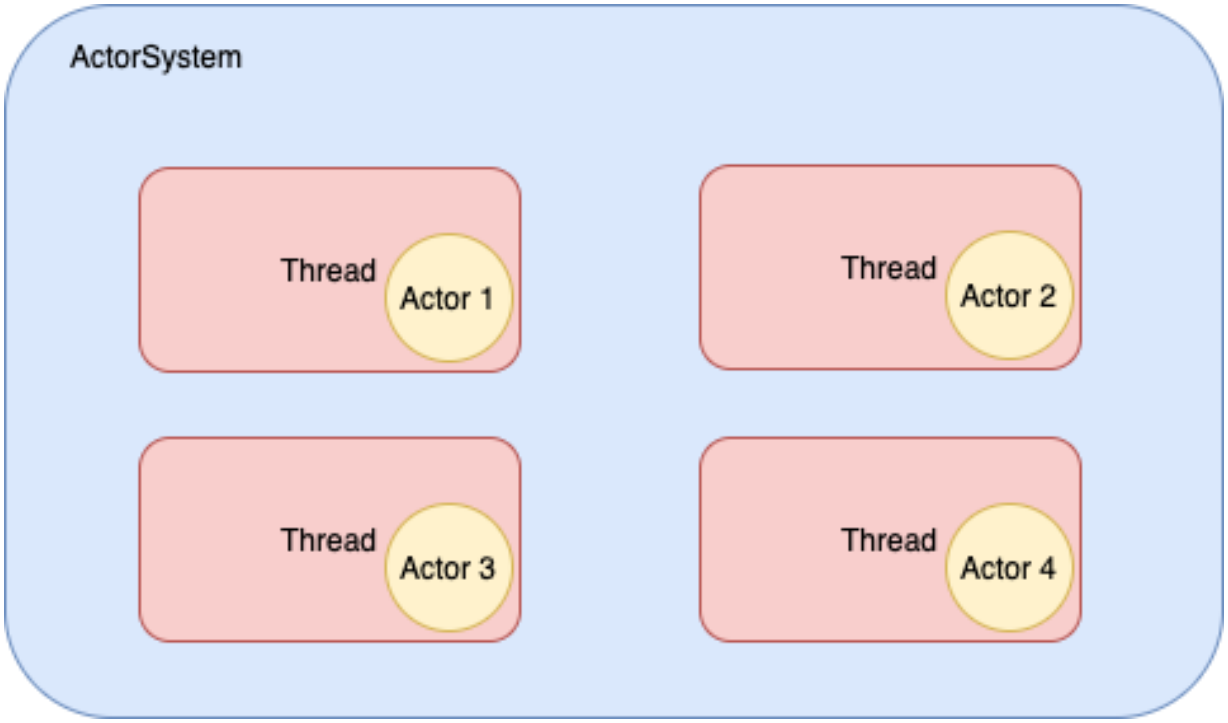



Figura 3.3: Organização de threads x atores no ActorSystem.

Assim, vamos construir o nosso ator da seguinte forma:

```
package com.casadocodigo.actors.robot

import akka.actor.typed.{ActorRef, Behavior, PostStop}
import akka.actor.typed.scaladsl.Behaviors

object CollectRobotActor {

  def apply(): Behavior[ComandoDeColeta] =
Behaviors.withStash(4) { buffer =>
    Behaviors.setup { contexto =>

      val refAtorDeTransmissao =
contexto.spawn(TransmitRobotActor(),
"TransmitRobotActor")

      Behaviors.receive[ComandoDeColeta] {
        (contexto, mensagem) =>
          mensagem match {
            case Coletar(_) =>
              if (buffer.isFull) {
                contexto.log.info(s"limite de coletas
atingido! Abortando operações!")

buffer.unstashAll(transmissor(refAtorDeTransmissao))
                Behaviors.stopped
              }
              else {
                buffer.stash(mensagem)
                contexto.log.info(s"amostra $mensagem
coletada com sucesso!")
                Behaviors.same
              }
            case IniciarTransmissao =>
```

```
buffer.unstashAll(transmissor(refAtorDeTransmissao))
              Behaviors.same
          }
      }.receiveSignal {
        case (context, PostStop) =>
          context.log.info("Módulo de coletas
encerrado. Favor reiniciar o módulo!")
          Behaviors.same
      }
    }
  }

  private def transmissor(refAtorDeTransmissao:
ActorRef[Transmitir]): Behavior[ComandoDeColeta] = {
    Behaviors.setup {
      _ =>
        Behaviors.receive {
          (contexto, mensagem) =>
            mensagem match {
              case Coletar(coleta) =>
                refAtorDeTransmissao !
Transmitir(coleta)
              case _ =>
                contexto.log.info(s"mensagem $mensagem
inválida!")
            }
            Behaviors.same
        }
    }
  }
}
```

Você pode observar, no método `transmissor`, que acrescentamos um case default (`case _`). O interpretador Scala executa checks em tempo de compilação que nos avisam se existem cenários não cobertos em nossos pattern matchings. Como nosso ator espera receber mensagens do tipo *ComandoDeColeta*, o interpretador está nos avisando que não estamos processando todos os tipos possíveis (no caso, o tipo que utilizamos para iniciar a transmissão). Assim, criamos um case default para satisfazer os checks do interpretador.

Nosso ator utiliza um *StashBuffer* de quatro posições. Temos dois tipos de mensagens, `Coletar e IniciarTransmissao`. Quando o ator recebe uma mensagem de coletar, o ator persiste a coleta no buffer, se possível. Caso o buffer esteja cheio, forçamos a parada do ator. Por fim, se o ator receber uma mensagem de iniciar transmissão, realizamos a transmissão de todas as coletas, esvaziando o buffer.

Atores possuem um ciclo de vida, no qual eles podem estar criados, ativos, finalizados, ou em processo de reinicialização. Vamos aprender isso com mais detalhes, mas, por ora, podemos entender que, quando um ator é parado, ele é removido do

ambiente de execução e não pode mais receber mensagens.

Observamos também uma função anônima, definida dentro de um método chamado `receiveSignal`. A função dentro do método define um *ator de ciclo de vida*. Atores de ciclo de vida não podem receber mensagens, apenas processam eventos do ciclo de vida do ator que estão observando. No decorrer do livro, aprenderemos mais sobre esses atores.

A seguir construímos o nosso ator de transmissão de análises. Esse é um ator bem simples, que simplesmente loga uma mensagem para simbolizar a transmissão:

```
package com.casadocodigo.actors.robot

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors

object TransmitRobotActor {

  def apply(): Behavior[Transmitir] = Behaviors.receive
{
    (contexto, mensagem) =>
      contexto.log.info(s"amostra $mensagem coletada
transmitida com sucesso para a base de operações!")
      Behaviors.same
  }

}
```

Vamos executar nosso modelo de atores para colocar o nosso robô em funcionamento. Modificamos o nosso `main`, inicializamos os módulos de movimentação e de coleta e enviamos alguns comandos para o nosso robô:

```
val atorMovimentoDoRobo =
system.systemActorOf(MoveRobotActor(),
"MoveRobotActor")
val atorColetorDoRobo =
system.systemActorOf(CollectRobotActor(),
"CollectRobotActor")
atorMovimentoDoRobo ! MoverParaFrente
atorMovimentoDoRobo ! MoverParaTras
atorMovimentoDoRobo ! MoverParaEsquerda
atorMovimentoDoRobo ! MoverParaFrente
atorMovimentoDoRobo ! MoverParaDireita

atorColetorDoRobo ! Coletar(Coleta(UUID.randomUUID(),
Random.nextDouble()))
atorColetorDoRobo ! Coletar(Coleta(UUID.randomUUID(),
Random.nextDouble()))
atorColetorDoRobo ! IniciarTransmissao
```

Ao executar, obtemos os seguintes resultados:

```
22:43:56.099 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a frente com sucesso!
22:43:56.100 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para trás com sucesso!
22:43:56.100 [MyActorSystem-akka.actor.default-
```

```
dispatcher-7] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a esquerda com sucesso!
22:43:56.100 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a frente com sucesso!
22:43:56.101 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a direita com sucesso!
22:43:56.104 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(f5a60837-afb6-43b0-9659-
6e07f6e725af,0.871130752045479)) coletada com sucesso!
22:43:56.104 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(e5e82bf0-886d-4c44-aa0f-
4c82ccf5b395,0.8649400951604682)) coletada com sucesso!
22:43:56.108 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(f5a60837-afb6-43b0-9659-
6e07f6e725af,0.871130752045479)) coletada transmitida
com sucesso para a base de operações!
22:43:56.108 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(e5e82bf0-886d-4c44-aa0f-
4c82ccf5b395,0.8649400951604682)) coletada transmitida
com sucesso para a base de operações!
```

Vamos agora observar o que acontece se tentarmos realizar mais do que quatro coletas. Vamos modificar

os comandos da seguinte forma:

```
atorMovimentoDoRobo ! MoverParaFrente
atorMovimentoDoRobo ! MoverParaTras
atorMovimentoDoRobo ! MoverParaEsquerda
atorMovimentoDoRobo ! MoverParaFrente
atorMovimentoDoRobo ! MoverParaDireita

atorColetorDoRobo ! Coletar(Coleta(UUID.randomUUID(),
Random.nextDouble()))
atorColetorDoRobo ! Coletar(Coleta(UUID.randomUUID(),
Random.nextDouble()))
atorColetorDoRobo ! Coletar(Coleta(UUID.randomUUID(),
Random.nextDouble()))
atorColetorDoRobo ! Coletar(Coleta(UUID.randomUUID(),
Random.nextDouble()))
atorColetorDoRobo ! Coletar(Coleta(UUID.randomUUID(),
Random.nextDouble()))
atorColetorDoRobo ! IniciarTransmissao
```

Ao executar esses comandos, veremos mensagens que indicam que a nossa tratativa funcionou, que todas as coletas foram transmitidas e que o ator foi finalizado. Além disso, receberemos um erro ao final, pois tentamos utilizar um ator que se encontra paralizado:

```
23:27:36.939 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a frente com sucesso!
23:27:36.939 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para trás com sucesso!
```

```
23:27:36.940 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a esquerda com sucesso!
23:27:36.940 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a frente com sucesso!
23:27:36.940 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(b76cf494-a46b-4a76-a468-
f74225b94a6d,0.09792217582680618)) coletada com
sucesso!
23:27:36.940 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.robot.MoveRobotActor$ - Robô
movido para a direita com sucesso!
23:27:36.941 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(e6b5542b-9d43-4144-bc92-
6505fe8949d5,0.8668740227234115)) coletada com sucesso!
23:27:36.941 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(6586cbdf-1f9f-4541-9952-
2991956c67ba,0.9112069681929461)) coletada com sucesso!
23:27:36.942 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(df471849-16ff-4a29-9d2e-
cd6064daabd3,0.20192180246028302)) coletada com
sucesso!
23:27:36.942 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
```

```
limite de coletas atingido! Abortando operações!
23:27:36.944 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(b76cf494-a46b-4a76-a468-
f74225b94a6d,0.09792217582680618)) coletada transmitida
com sucesso para a base de operações!
23:27:36.944 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(e6b5542b-9d43-4144-bc92-
6505fe8949d5,0.8668740227234115)) coletada transmitida
com sucesso para a base de operações!
23:27:36.945 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(6586cbdf-1f9f-4541-9952-
2991956c67ba,0.9112069681929461)) coletada transmitida
com sucesso para a base de operações!
23:27:36.945 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(df471849-16ff-4a29-9d2e-
cd6064daabd3,0.20192180246028302)) coletada transmitida
com sucesso para a base de operações!
23:27:36.949 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
Módulo de coletas encerrado. Favor reiniciar o módulo!
23:27:36.955 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO akka.actor.RepointableActorRef -
Message
[com.casadocodigo.actors.robot.IniciarTransmissao] to
Actor[akka://MyActorSystem/system/CollectRobotActor#661
279267] was not delivered. [1] dead letters
encountered. If this is not an expected behavior then
Actor[akka://MyActorSystem/system/CollectRobotActor#661
```

```
279267] may have terminated unexpectedly. This logging
can be turned off or adjusted with configuration
settings 'akka.log-dead-letters' and 'akka.log-dead-
letters-during-shutdown'.
```

Mais adiante, aprenderemos a configurar o sistema de atores para que ele reinicie para nós um ator parado automaticamente. Antes de abandonarmos o nosso robô, vamos desenvolver uma nova feature e aprender mais um recurso do Akka.

## 3.5 Entregando mensagens com atraso (delayed messages)

Após alguns dias de operação, os cientistas da base de operações perceberam que, por vezes, coletas eram feitas e esquecidas, ficando várias horas paradas antes de serem transmitidas. Isso resultava em problemas, pois os produtos químicos da análise ficavam "marinando" o solo no compartimento de coletas, resultando na corrosão das paredes do coletor a longo prazo.

Para evitar o problema de corrosão, é preciso garantir que a transmissão das coletas seja feita no máximo em até 2h após a coleta.

Dentro do contexto do Akka, podemos utilizar um *agendador (scheduler)*. Vamos modificar o nosso

ator de coleta para agendar uma transmissão 2 horas após a coleta, a cada vez que nós recebermos uma nova coleta. O agendador recebe uma referência para o próprio ator, junto da mensagem de iniciar a transmissão das coletas:

```
package com.casadocodigo.actors.robot

import akka.actor.typed.{ActorRef, Behavior, PostStop}
import akka.actor.typed.scaladsl.Behaviors

import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

object CollectRobotActor {

  def apply(): Behavior[ComandoDeColeta] =
Behaviors.withStash(4) { buffer =>
    Behaviors.setup { contexto =>

      val refAtorDeTransmissao =
contexto.spawn(TransmitRobotActor(),
"TransmitRobotActor")

      Behaviors.receive[ComandoDeColeta] {
        (contexto, mensagem) =>
          mensagem match {
            case Coletar(_) =>
              contexto.scheduleOnce(2 hours,
contexto.self, IniciarTransmissao)
              if (buffer.isFull) {
                contexto.log.info(s"limite de coletas
atingido! Abortando operações!")

buffer.unstashAll(transmissor(refAtorDeTransmissao))
```

```
                Behaviors.stopped
              }
              else {
                buffer.stash(mensagem)
                contexto.log.info(s"amostra $mensagem
coletada com sucesso!")
                Behaviors.same
              }
            case IniciarTransmissao =>

buffer.unstashAll(transmissor(refAtorDeTransmissao))
              Behaviors.same
          }
      }.receiveSignal {
        case (context, PostStop) =>
          context.log.info("Módulo de coletas
encerrado. Favor reiniciar o módulo!")
          Behaviors.same
      }
    }
  }

  private def transmissor(refAtorDeTransmissao:
ActorRef[Transmitir]): Behavior[ComandoDeColeta] = {
    Behaviors.setup {
      _ =>
        Behaviors.receive {
          (contexto, mensagem) =>
            mensagem match {
              case Coletar(coleta) =>
                refAtorDeTransmissao !
Transmitir(coleta)
              case _ =>
                contexto.log.info(s"mensagem $mensagem
inválida!")
            }
            Behaviors.same
```

```
        }
    }
  }
}
```

Ao executarmos o código, vemos a mensagem sendo entregue após o período de delay definido. Para que você não precise esperar tanto tempo, 2 horas, vamos remover a mensagem de transmissão do objeto `main` e colocar um delay de apenas cinco minutos:

```
**20:01:52.522** [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(f3ec1719-4d9d-48ac-832f-
5841311eefca,0.381092972614713)) coletada com sucesso!
**20:01:52.522** [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.robot.CollectRobotActor$ -
amostra Coletar(Coleta(0b8805d0-4d17-4be1-b5f8-
ce7e69c58a9d,0.8918848540011657)) coletada com sucesso!
**20:06:52.543** [MyActorSystem-akka.actor.default-
dispatcher-11] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(f3ec1719-4d9d-48ac-832f-
5841311eefca,0.381092972614713)) coletada transmitida
com sucesso para a base de operações!
**20:06:52.544** [MyActorSystem-akka.actor.default-
dispatcher-11] INFO
com.casadocodigo.actors.robot.TransmitRobotActor$ -
amostra Transmitir(Coleta(0b8805d0-4d17-4be1-b5f8-
ce7e69c58a9d,0.8918848540011657)) coletada transmitida
com sucesso para a base de operações!
```

E assim salvamos o módulo de coletas. Aprendemos muitas coisas neste capítulo, como o que são atores, como criá-los, enviar e processar múltiplos tipos de mensagens com um único ator, além do envio de mensagens com delay.

Este capítulo foi muito instrutivo e formou a base para os próximos capítulos que virão. Vamos deixar o nosso robô de lado agora e avançar para o próximo capítulo, onde aprenderemos mais sobre o ciclo de vida dos atores.

CAPÍTULO 4

# Ciclo de vida e tratativa de erros em atores

Anteriormente, mencionamos que os atores possuem um ciclo de vida. Vamos agora nos aprofundar nesses conceitos e os compreender com maior profundidade.

## 4.1 Estados de um ator

Um ator pode se encontrar em quatro status básicos: criado, reiniciado, ativo e finalizado (parado). Os status criado e reiniciado são de transição, a partir dos quais um ator segue para o status ativo. O status finalizado é um status terminal, a partir do qual o ator não pode mais ser utilizado. O status ativo é o único status onde é possível para um ator receber mensagens.

O diagrama a seguir demonstra os estados de um ator, junto com as transições de status:

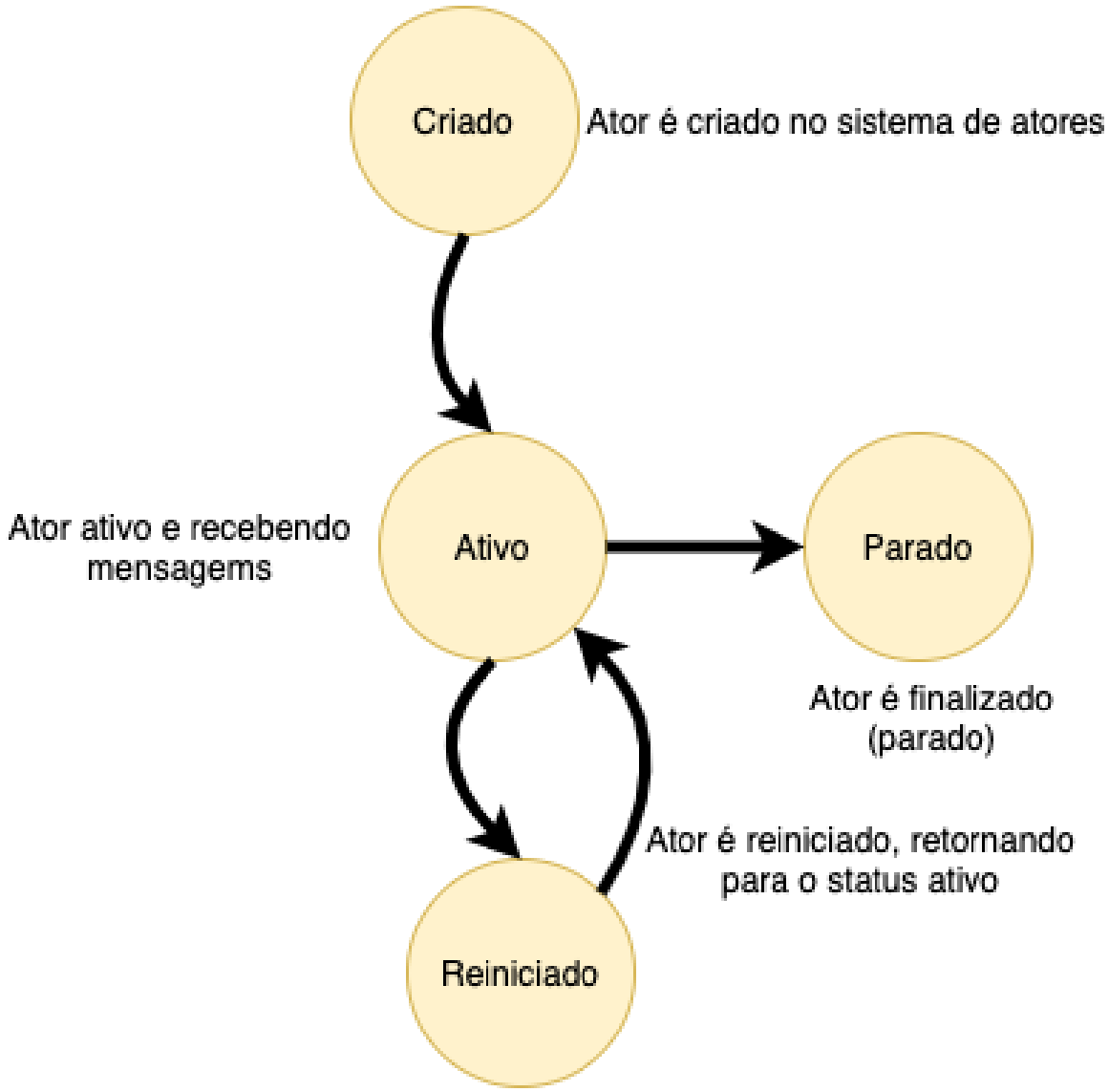



Figura 4.1: Máquina de estados de atores.

Anteriormente, vimos nosso ator ser finalizado utilizando o retorno `Behaviors.stopped` dentro da lógica de recebimento de mensagens. No nosso exemplo, era o próprio ator quem se finalizava automaticamente. Vamos ver uma variação dessa

técnica, onde temos outro ator induzindo a parada de um ator.

**Observação:** se você já tiver visto algo do Akka de versões mais antigas, pode ter se deparado com o cenário de um ator recebendo uma mensagem especial chamada *PoisonPill*. Esse método foi depreciado em versões mais recentes do framework, por isso ele não será demonstrado na prática aqui.

> Sim, isso mesmo, a mensagem se chama *PoisonPill* (pílula de veneno)! Na terminologia do Akka, diversos termos fazem alusões aos atores como se fossem pessoas realizando tarefas. Assim, é possível achar o sistema de atores sendo chamado de *stage* (palco), fazendo alusão a um palco onde atores atuam, e *reencarnação* como um sinônimo da reinicialização dos atores.

Vamos começar criando um ator que pode receber uma chamada de parada:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors

object StoppableActor {

  trait Mensagem
```

```
  case class MensagemProcessamento(dado: String)
extends Mensagem

  case class MensagemFinalizar() extends Mensagem

  def apply(): Behavior[Mensagem] = Behaviors.receive {
    (contexto, mensagem) =>
      mensagem match {
        case MensagemProcessamento(dado) =>
          contexto.log.info(s"processando a mensagem
$dado!")
          Behaviors.same
        case MensagemFinalizar() =>
          contexto.log.info(s"finalizando o
processamento!")
          Behaviors.stopped
      }
  }
}
```

Como podemos ver, trata-se de um ator bem simples. Temos simplesmente o estado do ator sendo modificado de acordo com o tipo da mensagem. Vamos agora codificar o outro ator, que ordenará a parada para o nosso primeiro ator:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors
import com.casadocodigo.actors.StoppableActor.
{MensagemFinalizar, MensagemProcessamento}

object CallerActor {
```

```
  trait MensagemChamadora

  case class MensagemSolicitarProcessamento(dado:
String) extends MensagemChamadora

  case class MensagemSolicitarFinalizacao() extends
MensagemChamadora

  def apply(): Behavior[MensagemChamadora] =
Behaviors.setup { contexto =>

    val refAtorParavel =
contexto.spawn(StoppableActor(), "StoppableActor")

    Behaviors.receive {
      (contexto, mensagem) =>
        mensagem match {
          case MensagemProcessamento(dado) =>
            contexto.log.info(s"solicitando o
processamento da mensagem $dado!")
            refAtorParavel !
MensagemProcessamento(dado)
            Behaviors.same
          case MensagemFinalizar() =>
            contexto.log.info(s"solicitando a
finalização do processamento!")
            refAtorParavel ! MensagemFinalizar()
            Behaviors.same
        }
    }
  }
}
```

Como é possível ver, também se trata de um ator bastante simples que serve como um simples encapsulador (*wrapper*) do outro ator.

Vamos fazer a seguinte modificação no objeto de setup do sistema de ator com o intuito de criar o ator que vamos utilizar no método `main` de nossa aplicação:

```
package com.casadocodigo.actors

import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{Behavior, SpawnProtocol,
Terminated}

object ActorSetup {

  def apply(): Behavior[SpawnProtocol.Command] =
Behaviors.setup {
    contexto =>

      contexto.spawn(CallerActor(), "CallerActor")

      Behaviors.receiveSignal[SpawnProtocol.Command] {
        case (_, Terminated(_)) =>
          Behaviors.stopped
      }

      SpawnProtocol()
  }
}
```

Vamos agora codificar o método principal e enviar algumas mensagens:

```
import akka.actor.typed.{ActorRef, ActorSystem, Props,
SpawnProtocol}
import akka.util.Timeout
import com.casadocodigo.actors.{ActorSetup,
```

```
CallerActor}
import com.casadocodigo.actors.CallerActor.
{MensagemChamadora, MensagemSolicitarFinalizacao,
MensagemSolicitarProcessamento}

import scala.concurrent.duration.DurationInt
import scala.concurrent.{ExecutionContext, Future}

object Application {

  def main(args: Array[String]): Unit = {

    import akka.actor.typed.scaladsl.AskPattern._
    implicit val system:
ActorSystem[SpawnProtocol.Command] =
ActorSystem(ActorSetup(), "MyActorSystem")
    implicit val ec: ExecutionContext =
system.executionContext
    implicit val timeout: Timeout = Timeout(3.seconds)

    val chamador: Future[ActorRef[MensagemChamadora]] =
{
      system.ask(SpawnProtocol.Spawn(behavior =
CallerActor(), name = "CallerActor", props =
Props.empty, _))
    }

    for (ref <- chamador) {
      ref ! MensagemSolicitarProcessamento("teste 1")
      ref ! MensagemSolicitarProcessamento("teste 2")
      ref ! MensagemSolicitarProcessamento("teste 3")
      ref ! MensagemSolicitarFinalizacao()
    }
  }
}
```

Antes de analisarmos os resultados da execução, vamos discutir os vários novos recursos que observamos no código.

Nesse exemplo, instanciamos nosso ator principal de um modo diferente. Até o momento, vínhamos utilizando o método `systemActorOf` para criar os atores que usávamos no método `main`. Porém, esse modelo de instanciamento foi depreciado pela comunidade: o modo atual de fazê-lo é o que vemos aqui, que passaremos a utilizar em todo o livro.

Para criar atores no novo padrão, devemos utilizar a função `setup`, que passamos como parâmetro no momento da criação do sistema de atores. Fizemos uma alteração no nosso setup e, além de criarmos os atores através do método `spawn`, também fazemos uma chamada a `SpawnProtocol` ao final da função. Essa chamada nos permite configurar o sistema de atores para fornecer referências para todos os atores criados dentro do setup.

Por fim, atualizamos o `main` para obter a referência ao nosso ator e enviamos algumas mensagens. Para referenciar o ator, solicitamos ao `SpawnProtocol` que obtenha para nós uma referência para o ator, através do método `ask`. Esse método é assíncrono, ou seja, a referência não é retornada imediatamente, sendo obtida apenas depois que o sistema de atores completa toda a sua inicialização.

Por isso, recebemos como retorno uma `Future`, que usamos para "escutar" quando o sistema ficar pronto e recebemos a referência ao ator para podermos trabalhar com ele. Por fim, usamos a referência e enviamos algumas mensagens.

Podemos observar outros dois recursos da linguagem Scala aqui: *implícitos* e *for comprehensions*. Implícitos nos permitem criar componentes que podem ser referenciados em qualquer ponto dentro do escopo em que o implícito foi criado. Por exemplo, quando criamos a variável *timeout*, ela passa a ser acessível por todo o código dentro do método `main`, incluindo código instanciado pelo método. Assim, se tivéssemos um método *A* definido como:

```
def A(param: String)(implicit timeout: Timeout) =
....
```

Dentro do método `main`, poderíamos simplesmente chamar o método assim:

```
...
A("teste")
...
```

E o parâmetro de *timeout* seria repassado para o método **implicitamente**. *Implícitos* é um recurso um pouco confuso às vezes mas muito poderoso, que vale aprender!

Outro recurso são as *for comprehensions*. Basicamente, elas nos permitem definir sequências de comandos, que geram ao final um ou mais resultados. No nosso cenário, utilizamos

uma `for comprehension` para obter o resultado da nossa `Future`, utilizando-a em seguida para enviar as mensagens.

Após a execução, podemos observar as mensagens tanto do ator solicitante quanto do ator parável, indicando que nossos atores foram criados com sucesso:

```
19:47:34.407 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 1!
19:47:34.408 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 2!
19:47:34.408 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 1!
19:47:34.409 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 3!
19:47:34.409 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 2!
19:47:34.409 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando a finalização do processamento!
19:47:34.409 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 3!
19:47:34.410 [MyActorSystem-akka.actor.default-
```

```
dispatcher-5] INFO
com.casadocodigo.actors.StoppableActor$ - finalizando o
processamento!
```

O que acontece se tentarmos processar outra mensagem depois de pedirmos a finalização? Vamos acrescentar mais mensagens e descobrir:

```
...
      ref ! MensagemSolicitarProcessamento("teste 1")
      ref ! MensagemSolicitarProcessamento("teste 2")
      ref ! MensagemSolicitarProcessamento("teste 3")
      ref ! MensagemSolicitarFinalizacao()
      ref ! MensagemSolicitarProcessamento("teste 4")
...

19:52:29.298 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 1!
19:52:29.299 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 2!
19:52:29.299 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 1!
19:52:29.299 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 3!
19:52:29.299 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 2!
19:52:29.299 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando a finalização do processamento!
```

```
19:52:29.300 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 3!
19:52:29.300 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 4!
19:52:29.300 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - finalizando o
processamento!
19:52:29.310 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO akka.actor.LocalActorRef - Message
[com.casadocodigo.actors.StoppableActor$MensagemProcess
amento] to Actor[akka://MyActorSystem/user/CallerActor-
1/StoppableActor#452591072] was not delivered. [1] dead
letters encountered. If this is not an expected
behavior then
Actor[akka://MyActorSystem/user/CallerActor-
1/StoppableActor#452591072] may have terminated
unexpectedly. This logging can be turned off or
adjusted with configuration settings 'akka.log-dead-
letters' and 'akka.log-dead-letters-during-shutdown'
```

Parecido com o que vimos no exemplo do nosso robô, não foi possível processar a última mensagem, pois o ator se encontrava parado.

Mas afinal, como fazemos para reiniciar um ator? A forma mais fácil de fazer isso é delegando essa tarefa ao próprio Akka através de um *supervisor*. Antes de começarmos a conhecer supervisores, vamos só entender qual a diferença deste formato com relação ao próprio ator se autofinalizar:

- Quando um ator se autofinaliza, o consumo da fila é parado **imediatamente**. Nenhuma mensagem é consumida após a sua parada (obviamente, o consumo é retomado após uma reinicialização);
- Quando paramos um ator através de uma mensagem externa, o ator não é parado imediatamente, pois a mensagem de parada é apenas mais uma na fila. Sendo assim, todo o processamento que foi enviado antes da solicitação de parada ser feita será processada, pela garantia da ordem de processamento feita pelo Akka.

Qual é a melhor estratégia? Como tudo na vida, depende do caso. Cabe analisar as características do problema para definir qual o melhor caminho a se seguir.

Agora vamos prosseguir o nosso aprendizado conhecendo mais sobre supervisores.

## 4.2 Supervisores

Até o presente momento, todas as vezes em que tivemos atores sendo parados, esse tem sido o fim de nossos atores: não é possível enviar novas

mensagens para processamento para um ator que se encontra parado.

O que acontece se algum erro ocorre no processamento? Vamos fazer um teste, modificando nosso ator para lançar uma exceção quando receber uma mensagem:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors
import com.casadocodigo.actors.StoppableActor.
{MensagemFinalizar, MensagemProcessamento}

object CallerActor {

  trait MensagemChamadora

  case class MensagemSolicitarProcessamento(dado:
String) extends MensagemChamadora

  case class MensagemSolicitarFinalizacao() extends
MensagemChamadora

  def apply(): Behavior[MensagemChamadora] =
Behaviors.setup { contexto =>

    val refAtorParavel =
contexto.spawn(StoppableActor(), "StoppableActor")

    Behaviors.receive {
      (contexto, mensagem) =>
        mensagem match {
          case MensagemSolicitarProcessamento(dado) =>
```

```
            throw new RuntimeException("BOOM!")
            Behaviors.same
          case MensagemSolicitarFinalizacao() =>
            contexto.log.info(s"solicitando a
finalização do processamento!")
            refAtorParavel ! MensagemFinalizar()
            Behaviors.same
        }
    }
  }
}
```

Executando o código, veremos que apenas a primeira mensagem é lida e as outras mensagens caem na *DLQ*:

```
19:01:17.865 [MyActorSystem-akka.actor.default-
dispatcher-7] ERROR akka.actor.SupervisorStrategy -
BOOM!
java.lang.RuntimeException: BOOM!
        at
com.casadocodigo.actors.CallerActor$.$anonfun$apply$2(C
allerActor.scala:23)
...
(stack-trace reduzido para melhor visualização)
...
19:01:17.880 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO akka.actor.LocalActorRef - Message
[com.casadocodigo.actors.CallerActor$MensagemSolicitarP
rocessamento] to
Actor[akka://MyActorSystem/user/CallerActor-
1#-669938178] was not delivered. [1] dead letters
encountered. If this is not an expected behavior then
Actor[akka://MyActorSystem/user/CallerActor-
1#-669938178] may have terminated unexpectedly. This
logging can be turned off or adjusted with
```

```
configuration settings 'akka.log-dead-letters' and
'akka.log-dead-letters-during-shutdown'.
19:01:17.882 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO akka.actor.LocalActorRef - Message
[com.casadocodigo.actors.CallerActor$MensagemSolicitarP
rocessamento] to
Actor[akka://MyActorSystem/user/CallerActor-
1#-669938178] was not delivered. [2] dead letters
encountered. If this is not an expected behavior then
Actor[akka://MyActorSystem/user/CallerActor-
1#-669938178] may have terminated unexpectedly. This
logging can be turned off or adjusted with
configuration settings 'akka.log-dead-letters' and
'akka.log-dead-letters-during-shutdown'.
19:01:18.098 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO akka.actor.LocalActorRef - Message
[com.casadocodigo.actors.CallerActor$MensagemSolicitarF
inalizacao] to
Actor[akka://MyActorSystem/user/CallerActor-
1#-669938178] was not delivered. [3] dead letters
encountered. If this is not an expected behavior then
Actor[akka://MyActorSystem/user/CallerActor-
1#-669938178] may have terminated unexpectedly. This
logging can be turned off or adjusted with
configuration settings 'akka.log-dead-letters' and
'akka.log-dead-letters-during-shutdown'.
```

DLQ é um acrônimo que significa, em inglês, *DeadLetter queue* (algo como fila de mensagens perdidas, ou "mortas"). Esse popular padrão de desenvolvimento de integrações define que mensagens que tiveram problemas de processamento sejam colocadas em uma fila especial, de modo que elas possam ser reprocessadas futuramente. A vantagem dessa estratégia *versus* simplesmente manter as mensagens em ciclo infinito de retentativas de processamento ou as re-enfileirar na própria fila, é que tais estratégias podem ocasionar quedas e sobrecargas do sistema, no caso de mensagens que possuam erros irrecuperáveis. Mais adiante no livro, aprenderemos como processar as mensagens que caem nessa fila.

Como podemos observar, caso ocorra algum erro, isso também ocasiona a parada do ator. Esse é o procedimento padrão do Akka. Agora, como podemos modificar esse comportamento?

Para isso, temos os chamados *supervisores*. Quando criamos atores, o ator-pai, ou seja, o ator dentro do qual os atores foram criados, é o supervisor daqueles atores, sendo responsável por definir a estratégia de tratativa de erros para o seu grupo de atores.

Para os atores que são instanciados na criação do sistema de atores, temos um ator especial, implícito, chamado *ator guardião*, que podemos utilizar para fazer a configuração para esses atores. O diagrama a seguir demonstra como essa estrutura se constitui na prática:

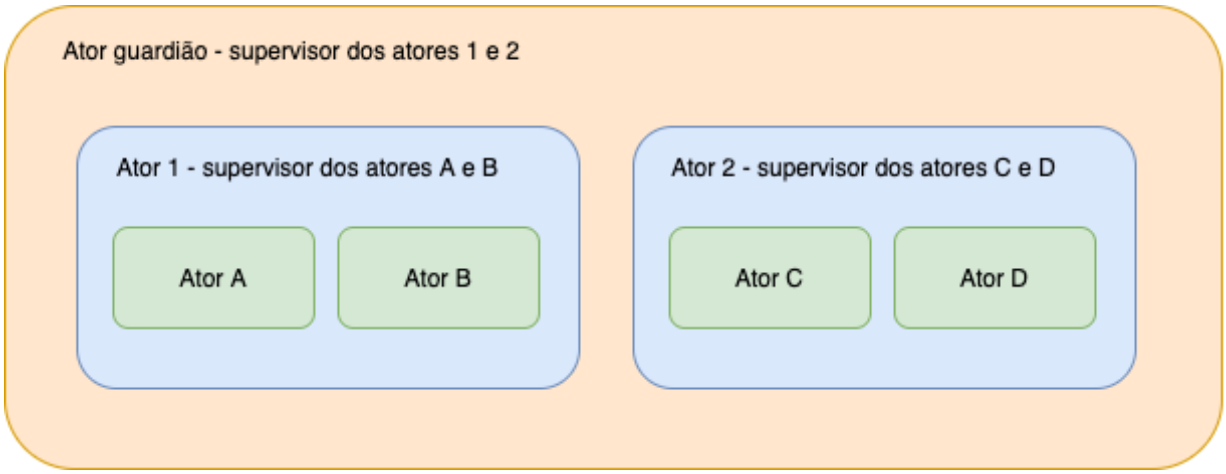

Ator guardião - supervisor dos atores 1 e 2
Ator 1 - supervisor dos atores A e B
Ator A
Ator B
Ator 2 - supervisor dos atores C e D
Ator C
Ator D

Figura 4.2: Hierarquia de supervisores x atores.

Estratégias de tratativas de erros se sobrepõem umas às outras. Isso significa que, se temos uma estratégia de *restart* (reinício) definida por um supervisor, por exemplo, e a estratégia padrão de parada definida no filho daquele supervisor - ele também, outro supervisor -, cada filho de cada supervisor seguirá a estratégia correspondente. O diagrama a seguir demonstra esse conceito na prática:

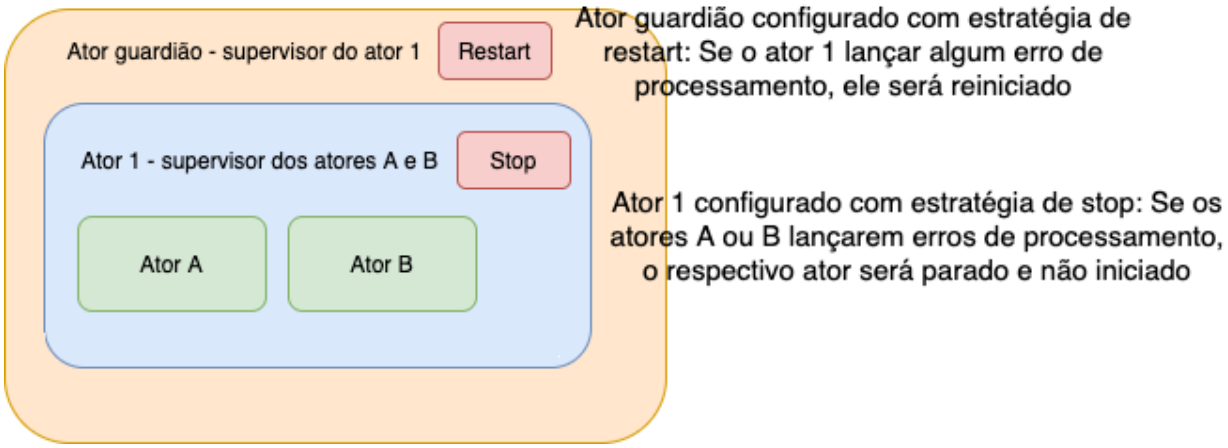

Ator guardião - supervisor do ator 1
Restart
Ator guardião configurado com estratégia de restart: Se o ator 1 lançar algum erro de processamento, ele será reiniciado
Ator 1 - supervisor dos atores A e B
Stop
Ator A
Ator B
Ator 1 configurado com estratégia de stop: Se os atores A ou B lançarem erros de processamento, o respectivo ator será parado e não iniciado

Figura 4.3: Hierarquia de estratégias de supervisores.

Agora, vamos experimentar fazer nossas primeiras configurações com supervisores. Vamos reutilizar o nosso exemplo anterior, acrescentando um supervisor que reinicie o nosso ator:

```
package com.casadocodigo.actors

import akka.actor.typed.{Behavior, SupervisorStrategy}
import akka.actor.typed.scaladsl.Behaviors

object StoppableActor {

  trait Mensagem

  case class MensagemProcessamento(dado: String)
extends Mensagem

  case class MensagemFinalizar() extends Mensagem

  case class ExcecaoDeFinalizacao(mensagem: String)
extends RuntimeException(mensagem)
```

```
  def apply(): Behavior[Mensagem] =
Behaviors.supervise[Mensagem](behavior())
    .onFailure[ExcecaoDeFinalizacao]
(SupervisorStrategy.restart)

  def behavior(): Behavior[Mensagem] =
Behaviors.receive {
    (contexto, mensagem) =>
      mensagem match {
        case MensagemProcessamento(dado) =>
          contexto.log.info(s"processando a mensagem
$dado!")
          Behaviors.same
        case MensagemFinalizar() =>
          contexto.log.info(s"finalizando o
processamento!")
          throw ExcecaoDeFinalizacao("Finalizando!")
      }
  }
}
```

Como podemos ver, é bastante simples configurar o supervisor para fazer um restart para qualquer erro ou problema que ocorra nos nossos atores, basta "envelopar" a sua definição com o método decorador `Behaviors.supervise` . Outra alteração é modificarmos o nosso ator parável, para que ele lance uma exceção, em vez de retornar um `Behaviors.stopped` .

Vamos executar novamente o código, utilizando a última versão que fizemos do método `main` , e observar que, desta vez, não temos nenhuma

mensagem enviada para a DLQ, e sim todas as mensagens processadas, por conta de nosso ator ter reiniciado:

```
20:00:45.382 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 1!
20:00:45.383 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 2!
20:00:45.383 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 3!
20:00:45.383 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 1!
20:00:45.383 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando a finalização do processamento!
20:00:45.384 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 2!
20:00:45.384 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 3!
20:00:45.384 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 4!
20:00:45.385 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - finalizando o
processamento!
20:00:45.390 [MyActorSystem-akka.actor.default-
```

```
dispatcher-6] ERROR
com.casadocodigo.actors.StoppableActor$ - Supervisor
RestartSupervisor saw failure: Finalizando!
com.casadocodigo.actors.StoppableActor$ExcecaoDeFinaliz
acao: Finalizando!
        at
com.casadocodigo.actors.StoppableActor$.$anonfun$behavi
or$1(StoppableActor.scala:27)
        at
akka.actor.typed.internal.BehaviorImpl$ReceiveBehavior.
receive(BehaviorImpl.scala:137)
...
(stack-trace reduzido para melhor visualização)
...
20:00:45.393 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 4!
```

Quando um ator é finalizado sem o uso de exceções (a partir do retorno `Behaviors.stop`), infelizmente não é mais possível reiniciá-lo, apenas criar uma nova instância a partir do método *spawn*. Para cenários em que desejamos que o ator finalize o processamento, porém com a possibilidade de reiniciar, o padrão do Akka é fazer isso por intermédio de exceções, por isso fizemos a modificação na forma como o ator realiza a sua finalização.

No nosso exemplo, criamos uma nova exceção, `ExcecaoDeFinalizacao`. Diferentes estratégias

podem ser empregadas de acordo com o tipo da exceção lançada. Vamos modificar o nosso código incluindo outra exceção, acompanhada de uma nova cláusula de estratégia de supervisão:

```
package com.casadocodigo.actors

import akka.actor.typed.{Behavior, SupervisorStrategy}
import akka.actor.typed.scaladsl.Behaviors

object StoppableActor {

  trait Mensagem

  case class MensagemProcessamento(dado: String)
extends Mensagem

  case class MensagemFinalizar() extends Mensagem

  case class ExcecaoDeFinalizacao(mensagem: String)
extends RuntimeException(mensagem)
  case class ExcecaoDeProcessamento(mensagem: String)
extends RuntimeException(mensagem)

  def apply(): Behavior[Mensagem] =
Behaviors.supervise(Behaviors.supervise(behavior())
    .onFailure[ExcecaoDeProcessamento]
(SupervisorStrategy.resume))
    .onFailure[ExcecaoDeFinalizacao]
(SupervisorStrategy.restart)

  def behavior(): Behavior[Mensagem] =
Behaviors.receive {
    (contexto, mensagem) =>
      mensagem match {
        case MensagemProcessamento(dado) =>
```

```
          contexto.log.info(s"processando a mensagem
$dado!")
          ExcecaoDeProcessamento("Falha no
processamento!")
          Behaviors.same
        case MensagemFinalizar() =>
          contexto.log.info(s"finalizando o
processamento!")
          throw ExcecaoDeFinalizacao("Finalizando!")
      }
  }
}
```

Se executarmos, vamos ver que os logs parecem os mesmos de antes:

```
21:26:22.003 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 1!
21:26:22.004 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 2!
21:26:22.004 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 3!
21:26:22.004 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando a finalização do processamento!
21:26:22.004 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 1!
21:26:22.005 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 4!
...
```

Isso se deve a estarmos usando a estratégia *resume*. Na estratégia resume (resumir), indicamos que o ator simplesmente continue o processamento sem reiniciar, "engolindo" a exceção e seguindo adiante com o processamento. Por isso, não vemos nenhuma diferença aparente quando lançamos a exceção de processamento.

Sumarizando, temos os seguintes tipos de estratégias disponíveis para supervisores:

- **resume**: nessa estratégia, o ator não é reiniciado, prosseguindo para a próxima mensagem;
- **restart**: nessa estratégia, o ator é reiniciado antes de prosseguir. Tal ação é útil se precisamos liberar recursos antes de prosseguir com o processamento, como recriar conexões com o banco de dados. Veremos mais sobre ações de liberação de recursos durante reinicializações quando falarmos sobre atores de ciclo de vida;
- **stop**: nessa estratégia, o ator é finalizado, não podendo mais ser inicializado, apenas recriado manualmente. Essa é a estratégia padrão do Akka.

Antes de prosseguirmos para a próxima seção, vamos aprender sobre políticas de retentativas

*(backoff policies)*. No nosso exemplo, nosso ator será reiniciado imediatamente após um erro ocorrer.

Se quisermos mudar esse comportamento, por exemplo, definindo que nosso ator reinicie em no mínimo um segundo, escalando para um máximo de um minuto de intervalo, em intervalos de retentativas aleatórias (com fator multiplicador de 0,5), modificamos o código da seguinte forma:

```
package com.casadocodigo.actors

import akka.actor.typed.{Behavior, SupervisorStrategy}
import akka.actor.typed.scaladsl.Behaviors

import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

object StoppableActor {

  trait Mensagem

  case class MensagemProcessamento(dado: String)
extends Mensagem

  case class MensagemFinalizar() extends Mensagem

  case class ExcecaoDeFinalizacao(mensagem: String)
extends RuntimeException(mensagem)

  case class ExcecaoDeProcessamento(mensagem: String)
extends RuntimeException(mensagem)

  def apply(): Behavior[Mensagem] =
Behaviors.supervise(Behaviors.supervise(behavior())
```

```
    .onFailure[ExcecaoDeProcessamento]
(SupervisorStrategy.resume))
    .onFailure[ExcecaoDeFinalizacao]
(SupervisorStrategy.restartWithBackoff(1 second, 1
minute, 0.5))

  def behavior(): Behavior[Mensagem] =
Behaviors.receive {
    (contexto, mensagem) =>
      mensagem match {
        case MensagemProcessamento(dado) =>
          contexto.log.info(s"processando a mensagem
$dado!")
          ExcecaoDeProcessamento("Falha no
processamento!")
          Behaviors.same
        case MensagemFinalizar() =>
          contexto.log.info(s"finalizando o
processamento!")
          throw ExcecaoDeFinalizacao("Finalizando!")
      }
  }
}
```

Em seguida, modificamos nosso método `main` para enviar várias mensagens e finalizações, forçando vários restarts:

```
      ref ! MensagemSolicitarProcessamento("teste 1")
      ref ! MensagemSolicitarProcessamento("teste 2")
      ref ! MensagemSolicitarProcessamento("teste 3")
      ref ! MensagemSolicitarFinalizacao()
      ref ! MensagemSolicitarProcessamento("teste 4")
      ref ! MensagemSolicitarProcessamento("teste 5")
      ref ! MensagemSolicitarFinalizacao()
      ref ! MensagemSolicitarProcessamento("teste 6")
```

```
        ref ! MensagemSolicitarProcessamento("teste 7")
        ref ! MensagemSolicitarFinalizacao()
        ref ! MensagemSolicitarProcessamento("teste 8")
        ref ! MensagemSolicitarProcessamento("teste 9")
        ref ! MensagemSolicitarProcessamento("teste 10")
```

Ao executar, podemos observar que algumas mensagens foram perdidas. Isso é um ponto importante a ser considerado se configurarmos uma política de retentativas no supervisor: quando a política é configurada, todas as mensagens enviadas durante o período em que o ator está esperando para ser reinicializado são perdidas:

```
...
22:00:07.867 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 8!
22:00:07.868 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 9!
22:00:07.868 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 10!
22:00:07.871 [MyActorSystem-akka.actor.default-
dispatcher-3] ERROR
com.casadocodigo.actors.StoppableActor$ - Supervisor
RestartSupervisor saw failure: Finalizando!
com.casadocodigo.actors.StoppableActor$ExcecaoDeFinaliz
acao: Finalizando!
        at
com.casadocodigo.actors.StoppableActor$.$anonfun$behavi
or$1(StoppableActor.scala:34)
        at
akka.actor.typed.internal.BehaviorImpl$ReceiveBehavior.
```

```
receive(BehaviorImpl.scala:137)
        at
akka.actor.typed.Behavior$.interpret(Behavior.scala:274
)
        at
akka.actor.typed.Behavior$.interpretMessage(Behavior.sc
ala:230)
        at
akka.actor.typed.internal.InterceptorImpl$$anon$2.apply
(InterceptorImpl.scala:57)
...
(stack-trace reduzido para melhor visualização)
...
22:00:09.175 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 4!
22:00:09.176 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 5!
22:00:09.176 [MyActorSystem-akka.actor.default-
dispatcher-5] INFO
com.casadocodigo.actors.StoppableActor$ - finalizando o
processamento!
22:00:09.191 [MyActorSystem-akka.actor.default-
dispatcher-5] ERROR
com.casadocodigo.actors.StoppableActor$ - Supervisor
RestartSupervisor saw failure: Finalizando!
...
```

Um último ponto que devemos aprender sobre supervisores é o comportamento dos atores filhos do ator supervisionado se configurarmos a estratégia de reiniciar. Quando um ator supervisionado é

reiniciado, todos os atores filhos daquele ator são reiniciados também. Vamos modificar o nosso exemplo para demonstrar isso.

Criaremos um novo ator, que será declarado como filho do ator supervisionado. Vamos configurar também um log no evento de parada do ator a fim de observar se ele reiniciou ou não:

```
package com.casadocodigo.actors

import akka.actor.typed.{Behavior, PostStop}
import akka.actor.typed.scaladsl.Behaviors

object ChildrenStoppableActor {

  case class OutraMensagemProcessamento(dado: String)

  def apply(): Behavior[OutraMensagemProcessamento] =
    Behaviors.receive[OutraMensagemProcessamento] {
      (contexto, mensagem) =>
        mensagem match {
          case OutraMensagemProcessamento(dado) =>
            contexto.log.info(s"processando a mensagem
$dado (filho)!")
            Behaviors.same
        }
    }.receiveSignal {
      case (context, PostStop) =>
        context.log.info("Ator filho sendo
reiniciado!")
        Behaviors.same
    }
}
```

E vamos incluí-lo no ator supervisionado, formando a hierarquia:

```
package com.casadocodigo.actors

import akka.actor.typed.{Behavior, SupervisorStrategy}
import akka.actor.typed.scaladsl.Behaviors
import
com.casadocodigo.actors.ChildrenStoppableActor.OutraMen
sagemProcessamento

import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

object StoppableActor {

  trait Mensagem

  case class MensagemProcessamento(dado: String)
extends Mensagem

  case class MensagemFinalizar() extends Mensagem

  case class ExcecaoDeFinalizacao(mensagem: String)
extends RuntimeException(mensagem)

  case class ExcecaoDeProcessamento(mensagem: String)
extends RuntimeException(mensagem)

  def apply(): Behavior[Mensagem] =
Behaviors.supervise(Behaviors.supervise(behavior())
    .onFailure[ExcecaoDeProcessamento]
(SupervisorStrategy.resume))
    .onFailure[ExcecaoDeFinalizacao]
(SupervisorStrategy.restartWithBackoff(1 second, 1
minute, 0.5))
```

```
  def behavior(): Behavior[Mensagem] = Behaviors.setup
{ contexto =>

    val refAtorFilho =
contexto.spawn(ChildrenStoppableActor(),
"ChildrenStoppableActor")

    Behaviors.receive {
      (contexto, mensagem) =>
        mensagem match {
          case MensagemProcessamento(dado) =>
            contexto.log.info(s"processando a mensagem
$dado!")
            refAtorFilho !
OutraMensagemProcessamento(dado)
            ExcecaoDeProcessamento("Falha no
processamento!")
            Behaviors.same
          case MensagemFinalizar() =>
            contexto.log.info(s"finalizando o
processamento!")
            throw ExcecaoDeFinalizacao("Finalizando!")
        }
    }
  }
}
```

Vamos executar novamente o mesmo bloco de mensagens que enviamos antes:

```
      ref ! MensagemSolicitarProcessamento("teste 1")
      ref ! MensagemSolicitarProcessamento("teste 2")
      ref ! MensagemSolicitarProcessamento("teste 3")
      ref ! MensagemSolicitarFinalizacao()
      ref ! MensagemSolicitarProcessamento("teste 4")
```

```
        ref ! MensagemSolicitarProcessamento("teste 5")
        ref ! MensagemSolicitarFinalizacao()
        ref ! MensagemSolicitarProcessamento("teste 6")
        ref ! MensagemSolicitarProcessamento("teste 7")
        ref ! MensagemSolicitarFinalizacao()
        ref ! MensagemSolicitarProcessamento("teste 8")
        ref ! MensagemSolicitarProcessamento("teste 9")
        ref ! MensagemSolicitarProcessamento("teste 10")
```

Ao executarmos, podemos ver que o ator filho também está reiniciando, como pode ser observado por este trecho nos logs:

```
...
13:50:59.456 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 6!
13:50:59.456 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 7!
13:50:59.457 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando a finalização do processamento!
13:50:59.457 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 8!
13:50:59.458 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 9!
13:50:59.458 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 10!
13:50:59.461 [MyActorSystem-akka.actor.default-
dispatcher-3] ERROR
com.casadocodigo.actors.StoppableActor$ - Supervisor
RestartSupervisor saw failure: Finalizando!
```

```
com.casadocodigo.actors.StoppableActor$ExcecaoDeFinaliz
acao: Finalizando!
        at
com.casadocodigo.actors.StoppableActor$.$anonfun$behavi
or$2(StoppableActor.scala:40)
        at
akka.actor.typed.internal.BehaviorImpl$ReceiveBehavior.
receive(BehaviorImpl.scala:137)
        at
akka.actor.typed.Behavior$.interpret(Behavior.scala:274
)
        at
akka.actor.typed.Behavior$.interpretMessage(Behavior.sc
ala:230)
...
(stack-trace reduzido para melhor visualização)
...
13:50:59.581 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO
com.casadocodigo.actors.ChildrenStoppableActor$ - Ator
filho sendo reiniciado!
13:51:00.698 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 4!
13:51:00.698 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.StoppableActor$ - processando a
mensagem teste 5!
13:51:00.698 [MyActorSystem-akka.actor.default-
dispatcher-8] INFO
com.casadocodigo.actors.ChildrenStoppableActor$ -
processando a mensagem teste 4 (filho)!
...
```

Os atores filhos sempre seguem a mesma estratégia definida no ator pai: se tivermos definida a estratégia padrão, por exemplo, de parada do ator em caso de erros, os atores filhos também serão parados. Essa regra se aplica a todos os atores que forem criados a partir do ator que configuramos, ou seja, se nossos atores filhos também tiverem seus atores filhos, esses atores seguirão respeitando a mesma estratégia configurada.

Agora que aprendemos como utilizar supervisores, vamos aprender como introduzir lógica dentro do ciclo de vida do nosso sistema com atores de ciclo de vida.

## 4.3 Atores de ciclo de vida

Vamos agora aprender como criar atores de ciclo de vida. Conforme dito anteriormente, atores de ciclo de vida não podem receber mensagens, apenas processam eventos do ciclo de vida dos atores que estão observando. Vamos criar um ator que processe eventos do nosso ator parável:

```
package com.casadocodigo.actors
```

```
import akka.actor.typed.{Behavior, PostStop,
PreRestart, SupervisorStrategy, Terminated}
import akka.actor.typed.scaladsl.Behaviors

import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

object StoppableActor {

  trait Mensagem

  case class MensagemProcessamento(dado: String)
extends Mensagem

  case class MensagemFinalizar() extends Mensagem

  case class ExcecaoDeFinalizacao(mensagem: String)
extends RuntimeException(mensagem)

  case class ExcecaoDeProcessamento(mensagem: String)
extends RuntimeException(mensagem)

  def apply(): Behavior[Mensagem] =
Behaviors.supervise(Behaviors.supervise(behavior())
    .onFailure[ExcecaoDeProcessamento]
(SupervisorStrategy.resume))
    .onFailure[ExcecaoDeFinalizacao]
(SupervisorStrategy.restartWithBackoff(1 second, 1
minute, 0.5))

  def behavior(): Behavior[Mensagem] =
Behaviors.receive[Mensagem] {
    (contexto, mensagem) =>
      mensagem match {
        case MensagemProcessamento(dado) =>
          contexto.log.info(s"processando a mensagem
$dado!")
```

```
          ExcecaoDeProcessamento("Falha no
processamento!")
          Behaviors.same
        case MensagemFinalizar() =>
          contexto.log.info(s"finalizando o
processamento!")
          throw ExcecaoDeFinalizacao("Finalizando!")
      }
  }.receiveSignal {
    case (context, PostStop) =>
      context.log.info("Evento de pós-parada do ator!")
      Behaviors.same
    case (context, PreRestart) =>
      context.log.info("Evento de pré-restart do
ator!")
      Behaviors.same
  }
}
```

Nesse código, criamos um ator que recebe os eventos de pré-reinicio (imediatamente antes de o ator ser reiniciado) e de pós-parada (imediatamente após o ator ser parado). No nosso exemplo, estamos realizando simples logs, mas em sistemas reais este recurso poderoso nos permite implementar tarefas importantes, como liberar conexões antes de uma reinicialização, por exemplo, ou monitorar quando um ator é paralisado.

Ao executarmos, podemos ver os logs do nosso ator de ciclo de vida, como esperado:

```
...
15:07:42.906 [MyActorSystem-akka.actor.default-
```

```
dispatcher-6] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 10!
15:07:42.907 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.StoppableActor$ - Evento de
pré-restart do ator!
15:07:42.909 [MyActorSystem-akka.actor.default-
dispatcher-3] ERROR
com.casadocodigo.actors.StoppableActor$ - Supervisor
RestartSupervisor saw failure: Finalizando!
...
```

Reparou que não tivemos logs do evento de pós-parada? Isso ocorre porque, no caso da estratégia de reiniciar, o ator não chega a transitar para o estado de parado, por isso não recebemos o evento.

Vamos modificar a supervisão para que o ator pare em vez de reiniciar:

```
 ...
  def apply(): Behavior[Mensagem] =
Behaviors.supervise(Behaviors.supervise(behavior())
    .onFailure[ExcecaoDeProcessamento]
(SupervisorStrategy.stop))
    .onFailure[ExcecaoDeFinalizacao]
(SupervisorStrategy.stop)
 ...
```

Após a mudança e executando novamente, podemos ver o evento de pós-parada aparecendo nos nossos logs:

```
15:14:29.498 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 10!
15:14:29.499 [MyActorSystem-akka.actor.default-
dispatcher-7] ERROR
com.casadocodigo.actors.StoppableActor$ - Supervisor
StopSupervisor saw failure: Finalizando!
com.casadocodigo.actors.StoppableActor$ExcecaoDeFinaliz
acao: Finalizando!
        at
com.casadocodigo.actors.StoppableActor$.$anonfun$behavi
or$1(StoppableActor.scala:34)
        at
akka.actor.typed.internal.BehaviorImpl$ReceiveBehavior.
receive(BehaviorImpl.scala:137)
...
(stack-trace reduzido para melhor visualização)
...
15:14:29.546 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.StoppableActor$ - Evento de
pós-parada do ator!
15:14:29.554 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO akka.actor.LocalActorRef - Message
[com.casadocodigo.actors.StoppableActor$MensagemProcess
amento] to Actor[akka://MyActorSystem/user/CallerActor-
1/StoppableActor#-2091057350] was not delivered. [1]
dead letters encountered. If this is not an expected
behavior then
Actor[akka://MyActorSystem/user/CallerActor-
1/StoppableActor#-2091057350] may have terminated
unexpectedly. This logging can be turned off or
adjusted with configuration settings 'akka.log-dead-
letters' and 'akka.log-dead-letters-during-shutdown'.
```

Estamos na reta final do nosso capítulo. Vamos observar agora como processar aquelas mensagens que caem na DLQ, que vimos em alguns dos nossos exemplos.

## 4.4 DeadLetters

Mensagens enviadas para a DLQ dentro do Akka são encapsuladas dentro de uma mensagem especial do Akka chamada *DeadLetter*. Vamos ver como podemos processar essas mensagens.

Primeiramente, vamos criar um novo ator, que processará mensagens das DLQs:

```
package com.casadocodigo.actors

import akka.actor.DeadLetter
import akka.actor.typed.{Behavior, SupervisorStrategy}
import akka.actor.typed.scaladsl.Behaviors

object DeadLetterActor {

  def apply(): Behavior[DeadLetter] =
Behaviors.supervise(behavior())
    .onFailure(SupervisorStrategy.restart)

  def behavior(): Behavior[DeadLetter] =
Behaviors.receive {
    (contexto, mensagem) =>
      contexto.log.info(s"processando a mensagem
${mensagem.message} da DeadLetter!")
```

```
      Behaviors.same
  }
}
```

> Dentro dos padrões do Akka, nosso ator receberá **TODAS** as mensagens enviadas para as DLQs do framework. Para realizar processamentos diferentes de acordo com a DLQ, podemos utilizar o campo *message* que recebemos da mensagem de *DeadLetter*, processando de acordo com o seu tipo.

A seguir, vamos adicionar o instanciamento do ator ao nosso setup:

```
package com.casadocodigo.actors

import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{Behavior, SpawnProtocol,
Terminated}

object ActorSetup {

  def apply(): Behavior[SpawnProtocol.Command] =
Behaviors.setup {
    contexto =>

      contexto.spawn(CallerActor(), "CallerActor")
      contexto.spawn(DeadLetterActor(),
"DeadLetterActor")

      Behaviors.receiveSignal[SpawnProtocol.Command] {
        case (_, Terminated(_)) =>
```

```
            Behaviors.stopped
        }

        SpawnProtocol()
    }
}
```

Por fim, vamos associar o nosso ator ao processamento das DLQs no sistema de atores com a intenção de indicar que todas as mensagens devem ser entregues a ele:

```
import akka.actor.DeadLetter
import
akka.actor.typed.eventstream.EventStream.Subscribe
import akka.actor.typed.{ActorRef, ActorSystem, Props,
SpawnProtocol}
import akka.util.Timeout
import com.casadocodigo.actors.{ActorSetup,
CallerActor, DeadLetterActor}
import com.casadocodigo.actors.CallerActor.
{MensagemChamadora, MensagemSolicitarFinalizacao,
MensagemSolicitarProcessamento}

import scala.concurrent.duration.DurationInt
import scala.concurrent.{ExecutionContext, Future}

object Application {

  def main(args: Array[String]): Unit = {

    import akka.actor.typed.scaladsl.AskPattern._
    implicit val system:
ActorSystem[SpawnProtocol.Command] =
ActorSystem(ActorSetup(), "MyActorSystem")
    implicit val ec: ExecutionContext =
```

```
system.executionContext
    implicit val timeout: Timeout = Timeout(3.seconds)

    val chamador: Future[ActorRef[MensagemChamadora]] =
{
      system.ask(SpawnProtocol.Spawn(behavior =
CallerActor(), name = "CallerActor", props =
Props.empty, _))
    }
    val atorDeadLetter: Future[ActorRef[DeadLetter]] =
{
      system.ask(SpawnProtocol.Spawn(behavior =
DeadLetterActor(), name = "DeadLetterActor", props =
Props.empty, _))
    }

    for (ref <- chamador; refDL <- atorDeadLetter) {
      system.eventStream ! Subscribe(refDL)
      ref ! MensagemSolicitarProcessamento("teste 1")
      ref ! MensagemSolicitarProcessamento("teste 2")
      ref ! MensagemSolicitarProcessamento("teste 3")
      ref ! MensagemSolicitarFinalizacao()
      ref ! MensagemSolicitarProcessamento("teste 4")
      ref ! MensagemSolicitarProcessamento("teste 5")
      ref ! MensagemSolicitarFinalizacao()
      ref ! MensagemSolicitarProcessamento("teste 6")
      ref ! MensagemSolicitarProcessamento("teste 7")
      ref ! MensagemSolicitarFinalizacao()
      ref ! MensagemSolicitarProcessamento("teste 8")
      ref ! MensagemSolicitarProcessamento("teste 9")
      ref ! MensagemSolicitarProcessamento("teste 10")
    }
  }
}
```

Para fazer a associação, usamos o `eventStream` do objeto `system`, passando a referência para o ator. Por fim, vamos manter a mesma configuração que já fizemos na supervisão, de modo que, para qualquer exceção, o ator pare e as mensagens caiam na DLQ:

```
 ...
  def apply(): Behavior[Mensagem] =
Behaviors.supervise(Behaviors.supervise(behavior())
    .onFailure[ExcecaoDeProcessamento]
(SupervisorStrategy.stop))
    .onFailure[ExcecaoDeFinalizacao]
(SupervisorStrategy.stop)
 ...
```

Executando novamente nosso código, podemos ver as mensagens do ator que processa as DLQs mostrando que nossa codificação foi um sucesso:

```
...
16:06:31.992 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 9!
16:06:31.993 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO com.casadocodigo.actors.CallerActor$
- solicitando o processamento da mensagem teste 10!
16:06:31.995 [MyActorSystem-akka.actor.default-
dispatcher-7] ERROR
com.casadocodigo.actors.StoppableActor$ - Supervisor
StopSupervisor saw failure: Finalizando!
com.casadocodigo.actors.StoppableActor$ExcecaoDeFinaliz
acao: Finalizando!
        at
```

```
com.casadocodigo.actors.StoppableActor$.$anonfun$behavi
or$1(StoppableActor.scala:33)
        at
akka.actor.typed.internal.BehaviorImpl$ReceiveBehavior.
receive(BehaviorImpl.scala:137)
...
(stack-trace reduzido para melhor visualização)
...
16:06:32.004 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO
com.casadocodigo.actors.StoppableActor$ - Evento de
pós-parada do ator!
16:06:32.039 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.DeadLetterActor$ - processando
a mensagem MensagemProcessamento(teste 4) da
DeadLetter!
16:06:32.040 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.DeadLetterActor$ - processando
a mensagem MensagemProcessamento(teste 5) da
DeadLetter!
16:06:32.040 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.DeadLetterActor$ - processando
a mensagem MensagemFinalizar() da DeadLetter!
16:06:32.040 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.DeadLetterActor$ - processando
a mensagem MensagemProcessamento(teste 6) da
DeadLetter!
16:06:32.041 [MyActorSystem-akka.actor.default-
dispatcher-3] INFO
com.casadocodigo.actors.DeadLetterActor$ - processando
a mensagem MensagemProcessamento(teste 7) da
DeadLetter!
16:06:32.041 [MyActorSystem-akka.actor.default-
```

```
dispatcher-3] INFO
com.casadocodigo.actors.DeadLetterActor$ - processando
a mensagem MensagemFinalizar() da DeadLetter!
...
```

E assim concluímos nosso estudo sobre tratativa de erros e ciclo de vida de atores no Akka. Podemos acompanhar a configuração de diversas estratégias de supervisão que garantem que o nosso código seja bastante robusto e se recupere automaticamente de falhas.

Agora, vamos voltar um pouco no tempo, ao nosso exemplo do capítulo 1, em que nosso e-commerce teve uma triste Black Friday. Vamos aprender agora como construir uma API em Akka, mais bem preparada para os problemas de sobrecarga que vimos naquela história.

## Capítulo 5

# Construindo uma API: parte 1 - criando a persistência e serviços

Agora que já temos uma boa base do framework Akka e do modelo de atores, vamos desenvolver uma API REST funcional. Nossa API fará operações de manipulação de produtos utilizando um servidor de mocks para simular uma API de estoque (sim, vamos simular a API de e-commerce do capítulo 1!).

Para desenvolver nossa API, utilizaremos o Akka HTTP, projeto dentro do framework responsável por fornecer uma solução robusta e de fácil utilização para construir aplicações HTTP.

Nossa aplicação será construída em duas partes. Neste primeiro capítulo, focaremos na construção das camadas mais internas da API, a de persistência com a base de dados e a de serviço. No capítulo seguinte, construiremos a camada responsável por expor os endpoints da API e realizaremos alguns testes utilizando uma biblioteca para testes de carga.

## 5.1 Montando o ambiente

Vamos começar adicionando as dependências. Vamos modificar nosso arquivo `.sbt` para adicionar todas as dependências que nossa API utilizará:

```
name := "akka-api"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"
val akkaHttpVersion = "10.2.6"
val slickVersion = "3.3.3"

dockerExposedPorts ++= Seq(8080)

enablePlugins(JavaAppPackaging)

mainClass in Compile := Some("com.casadocodigo.Boot")

//Akka e outras dependencias
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test,
  "com.typesafe.akka" %% "akka-http" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-spray-json" %
akkaHttpVersion
)

// dependencias de logging
```

```
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)

// dependencias de banco
libraryDependencies ++= Seq(
  "com.typesafe.slick" %% "slick" % slickVersion,
  "com.typesafe.slick" %% "slick-hikaricp" %
slickVersion,
  "org.postgresql" % "postgresql" % "42.2.23"
)
```

No nosso exemplo, adicionamos as seguintes dependências, além das que já utilizávamos anteriormente:

- **akka-http**: biblioteca do Akka responsável por nos fornecer um servidor HTTP junto à estrutura necessária para construir a camada de comunicação (REST) da nossa API.
- **akka-streams**: aqui temos o nosso primeiro contato com o Akka Streams! Ela é uma das dependências necessárias para utilizar o Akka HTTP. Aprenderemos mais sobre ela nos próximos capítulos, mas, neste momento, basta saber que ela é utilizada internamente pelo Akka HTTP para fazer o seu trabalho.
- **slick**: biblioteca que permite a implementação de acesso a bases de dados relacionais através da manipulação de objetos, sem a necessidade de pesadas codificações para acesso ao banco.

- **slick-hikaricp**: biblioteca de integração entre o Slick e o Hikari. A biblioteca Hikari é um sistema de controle de conexões (*connection pool*) que estamos adicionando para que ele seja responsável por gerenciar as nossas conexões de banco através do Slick;
- **spray-json**: biblioteca que permite configurar objetos para serem serializados/deserializados do formato JSON para objetos, assim podemos implementar a interface de nossa API de forma transparente.
- **sbt native packager**: plugin sbt que gera pacotes nativos ( `.jar` , `.zip` etc.). Utilizaremos esse plugin para empacotar a aplicação dentro de uma imagem Docker.

Para configurar o plugin `sbt native packager` , além de mexer no arquivo `build.sbt` , também é necessário incluir um arquivo chamado `plugins.sbt` dentro da pasta `project` com o seguinte comando:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.8.0")
```

A base de dados da nossa API será em Postgres. Vamos utilizar a tecnologia Docker de conteinerização para criar a base de dados. Também

criaremos uma imagem da aplicação, de modo que consigamos rodar a API com um simples `shell script`.

O shell script é muito simples, sendo simplesmente um comando para o native packager publicar uma imagem local e iniciar o ambiente com o docker-compose:

```
#!/usr/bin/env bash

sbt docker:publishLocal

docker-compose up -d
```

> Docker-compose é uma ferramenta do conjunto de ferramentas da Docker que permite criarmos vários contêineres com tudo o que é necessário para a nossa aplicação, além da própria aplicação. Para saber mais sobre o docker-compose, acesse: https://docs.docker.com/compose/.

Continuando a montagem do nosso ambiente, vamos adicionar um arquivo chamado `docker-compose.yml`. Esse é o arquivo que o comando `docker-compose up -d` vai procurar para configurar o nosso ambiente:

```
version: '3.9'
services:
  db:
    image: postgres
    ports:
      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: teste
      POSTGRES_DB: ecommerce
  api:
    image: akka-api:1.0
    tty: true
    ports:
      - "8080:8080"
```

Agora, vamos começar a codificação propriamente dita. Vamos criar um objeto Scala chamado `Boot` dentro do pacote `com.casadocodigo`. Nesse objeto, vamos instanciar o sistema de atores, além do servidor HTTP, que escutará no endereço *0.0.0.0:8080*, conforme configuramos no docker-compose:

```
package com.casadocodigo

import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.http.scaladsl.server.Directives._

import scala.concurrent.ExecutionContextExecutor
import scala.io.StdIn


object Boot extends App {
```

```
  implicit val system: ActorSystem =
ActorSystem("system")
  implicit val executionContext:
ExecutionContextExecutor = system.dispatcher

  val route =
    path("hello") {
      get {

complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "
<h1>Olá, Akka!</h1>"))
      }
    }

  val bindingFuture = Http().newServerAt("0.0.0.0",
8080).bind(route)

  StdIn.readLine()
  bindingFuture
    .flatMap(_.unbind())
    .onComplete(_ => system.terminate())
}
```

É possível notar que criamos um implícito chamado `executionContext`. O `executionContext` consiste em um objeto de acesso ao sistema de atores, que é passado implicitamente para o HTTP server linhas abaixo (o simples instanciamento dos implícitos já é suficiente, sem a necessidade de passá-los explicitamente no código). Esse objeto é usado pelo Akka HTTP para criar os seus atores internos.

Os `executionContext` nos permitem configurar como o nosso sistema de atores vai trabalhar com threads - configurando coisas como o pool de threads utilizado ou se apenas uma thread será utilizada -, quantidade máxima de threads que o pool poderá crescer, fator de multiplicação do thread pool, indicando, por exemplo, que queremos que o nosso thread pool escale em blocos de 10 em 10 threads, além da quantidade máxima de mensagens que uma thread deve processar de um mesmo ator antes de alternar o processamento para outro ator. Tipicamente as configurações padrões nos atendem muito bem e não há a necessidade de mexer nas configurações desse objeto, mas é importante ter conhecimento da sua função dentro do Akka.

Além de instanciar o servidor, criamos uma **rota** no nosso código. No Akka HTTP, implementamos a nossa camada de comunicação através de rotas, que são passadas para o servidor através do método `bind`. No nosso primeiro exemplo, configuramos uma rota HTTP GET, de caminho `/hello`, que retornará o texto **Olá, Akka!**.

## 5.2 Fazendo a nossa primeira chamada HTTP

Vamos executar a aplicação rodando o script:

```
./run_application.sh
```

O script realizará uma série de operações, como baixar imagens Docker, construir a imagem da nossa API e subir o nosso banco de dados. Ao final, ele subirá os contêineres do docker-compose em *background* e será encerrado com sucesso, como pode ser visto no fragmento a seguir:

```
[info] Built image akka-api with tags [1.0]
[success] Total time: 5 s, completed Sep 16, 2021,
7:55:36 PM
capitulo_5_6_db_1 is up-to-date
capitulo_5_6_api_1 is up-to-date
```

Executando o comando `docker ps` no terminal, podemos ver que tanto o banco quanto a nossa API

estão em execução (não se preocupe, em breve vamos utilizar o nosso banco):

```
   docker ps
CONTAINER ID   IMAGE          COMMAND
CREATED         STATUS         PORTS
NAMES
5ca863c33cf5   postgres       "docker-entrypoint.s…"
4 minutes ago   Up 4 minutes   0.0.0.0:5432->5432/tcp
capitulo_5_6_db_1
6f9dcaf6e878   akka-api:1.0   "/opt/docker/bin/akk…"
4 minutes ago   Up 4 minutes   0.0.0.0:8080->8080/tcp
capitulo_5_6_api_1
```

Vamos testar nosso endpoint? Para isso, vamos usar o bom e velho `curl` e popular o programa utilizado para fazer chamadas REST via terminal.

> O uso do `curl` é apenas uma sugestão. Se você desejar utilizar uma interface gráfica, sugiro utilizar o Postman: https://www.postman.com/

Vamos fazer a seguinte chamada no `curl` :

```
curl -X GET http://localhost:8080/hello
```

E no terminal veremos o nosso **Olá, Akka!** aparecendo:

```
#( 08/10/21@ 8:35PM )( alexandrelourenco@Alexandres-MacBook-Pro ):~
   curl -X GET http://localhost:8080/hello
<h1>Olá, Akka!</h1>%
#( 08/10/21@ 8:36PM )( alexandrelourenco@Alexandres-MacBook-Pro ):~
```

Figura 5.1: Resultado da nossa primeira chamada HTTP.

Também é possível realizar essa chamada simplesmente jogando a URL no seu navegador de preferência. Assim, podemos ver uma mensagem mais amigável:

Figura 5.2: Resultado da nossa primeira chamada HTTP no navegador.

Agora que temos o nosso ambiente devidamente montado e configurado, vamos começar a montar a nossa aplicação!

## 5.3 Modelando o acesso a banco de dados da nossa API

Agora que já temos o nosso esqueleto básico do ambiente da nossa API, vamos começar a construir a camada responsável pelo acesso ao banco de dados.

Nossa API será um módulo de vendas de um e-commerce, como descrito no capítulo 1. Ela compreende a gestão de pedidos e clientes, bem como o catálogo de produtos. O estoque é controlado por outra API, que será simulada através de um servidor de mocks.

O diagrama a seguir compreende o modelo entidade-relacionamento da nossa aplicação:

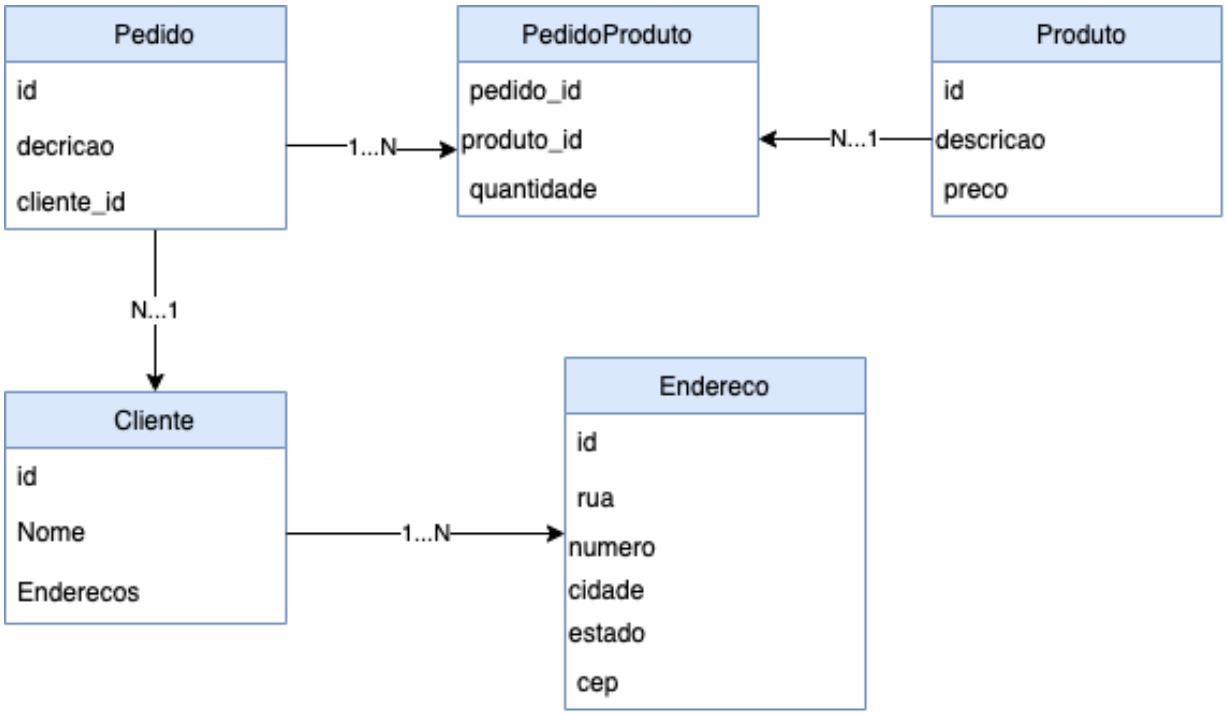



Figura 5.3: Modelo de entidades da aplicação.

Agora, vamos transportar esse modelo para a nossa aplicação. Primeiramente, vamos fazer as configurações para utilizar o banco de dados. Nós vamos incluir na aplicação o suporte a configurações dinâmicas, de modo que nossa aplicação possa se conectar ao banco, seja executando dentro do contêiner, seja sendo executada diretamente via sbt.

Vamos começar adicionando mais algumas variáveis no docker-compose:

```
version: '3.9'
services:
  db:
    image: postgres
```

```
    ports:
      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: teste
      POSTGRES_DB: ecommerce
  api:
    image: akka-api:1.0
    tty: true
    ports:
      - "8080:8080"
    environment:
      DB_HOST: db
      DB_PASS: teste
      ENVIRONMENT: prod
```

Incluímos no contêiner da aplicação três variáveis, uma com a senha do usuário do banco e outra com o endereço do banco. No docker-compose, é criada uma rede virtual entre os contêineres, o que facilita muito a conexão entre eles.

Nesse cenário, para que um contêiner acesse o outro, basta utilizar o nome do contêiner como o endereço de acesso. Por fim, criamos uma variável para identificar o próprio ambiente, no caso chaveando para o ambiente de produção (*prod*).

Em seguida, vamos criar os arquivos de configuração. Criar arquivos de configuração seguindo o padrão estabelecido pelo Scala é o necessário para começar a utilizar as configurações. Vamos começar criando um arquivo de

configurações base que todos os ambientes utilizarão:

```
# Datasources
database.postgres {
  connectionPool = "HikariCP"
  dataSourceClass =
"org.postgresql.ds.PGSimpleDataSource"
  properties = {
    serverName = "localhost"
    serverName = ${?DB_HOST}
    portNumber = "5432"
    databaseName = "ecommerce"
    user = "postgres"
    password = "teste"
    password = ${?DB_PASS}
  }
  numThreads = 10
}
```

Nesse arquivo, criamos as configurações do banco de dados. Algumas das configurações que inserimos possuem valores dinâmicos, como `${?DB_PASS}`. Isso significa que, para essa variável, por exemplo, se existir uma variável definida no sistema operacional com o nome `DB_PASS`, o valor dessa variável será capturado e utilizado pela aplicação, ou o valor `teste` será utilizado, caso a variável não exista.

Para o ambiente local, que roda sem utilizar o Docker, utilizaremos um ambiente chamado

*application*. O arquivo desse ambiente simplesmente importa o arquivo base, sem modificar nada:

```
include "application.base.conf"
```

Para o ambiente dockerizado, criaremos um ambiente chamado *prod*. Esse ambiente importará o arquivo base e modificará o *log level* do Akka para o nível `INFO` :

```
include "application.base.conf"

akka.loglevel = "INFO"
```

A imagem a seguir mostra a estrutura de arquivos de configuração da nossa aplicação:

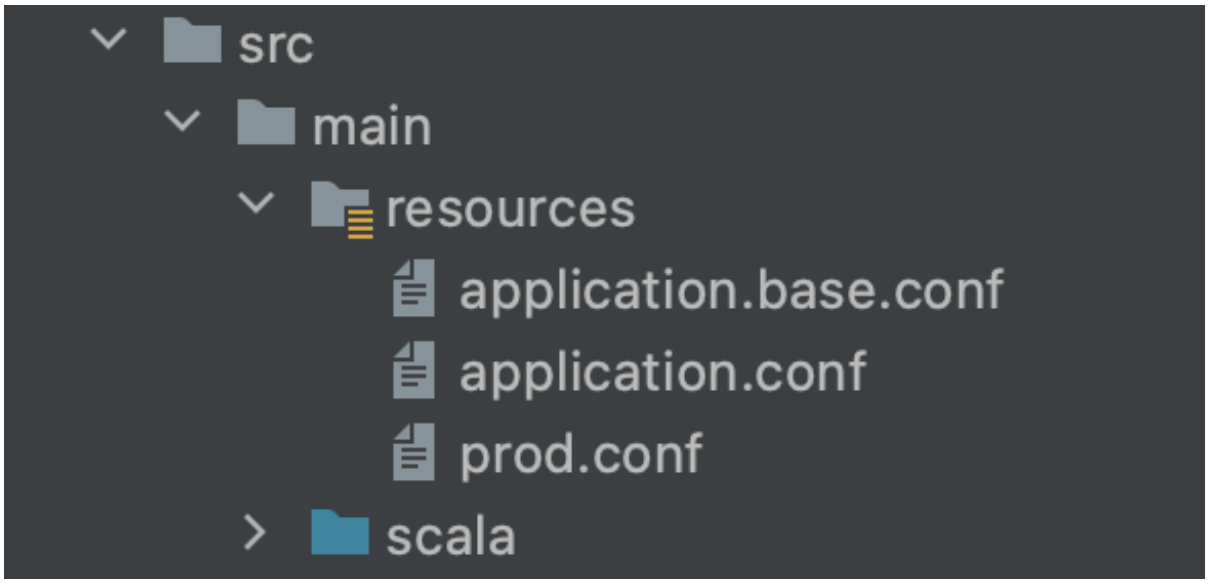



Figura 5.4: Hierarquia de arquivos de configuração da aplicação.

Configurações criadas, vamos fazer o código que as utilizará. Vamos começar criando uma *trait* chamada `DBConnection`. Essa trait será responsável por criar o pool de conexões com o banco (através do objeto `Database` do Slick), prover métodos utilitários para executar comandos e abrir streams com o banco de dados (vamos entender melhor como isso funciona na próxima seção):

```
package com.casadocodigo.repository

import com.typesafe.config.ConfigFactory
import slick.basic.DatabasePublisher
import slick.dbio.Effect.Read
import slick.jdbc.PostgresProfile.api._

import scala.concurrent.Future

trait DBConnection {
  val disableAutocommit =
SimpleDBIO(_.connection.setAutoCommit(false))

  def run[T](dBIOAction: DBIOAction[T, NoStream,
Nothing]): Future[T] = DBConnection.db.run(dBIOAction)

  def stream[T](dBIOAction: DBIOAction[Seq[T],
Streaming[T], Read]): DatabasePublisher[T] =
DBConnection.db.stream(disableAutocommit andThen
dBIOAction.withStatementParameters(fetchSize = 1000))
}

object DBConnection {
  lazy val db = Database.forConfig("database.postgres",
ConfigFactory.load(Option(
    System.getenv("ENVIRONMENT"))
```

```
.getOrElse(Option(System.getProperty("ENVIRONMENT"))
      .getOrElse("application"))))
}
```

Antes de prosseguirmos com a codificação, vamos só parar uns instantes para entender o básico dos conceitos do Slick.

## 5.4 Conceitos básicos do Slick e criando nossos repositórios

Conforme dito anteriormente, o Slick é um framework de acesso a bases de dados. Isso significa que criaremos nossas tabelas como classes na nossa aplicação. Essas classes estenderão a infraestrutura do framework, estabelecendo a ligação entre os objetos e as tabelas.

Esses objetos representando as tabelas são então utilizados para criar instâncias de um tipo de objeto especial do Slick chamado `TableQuery`. Esse objeto permite a criação de objetos do tipo `DBIOAction`, que representam operações de banco de dados, como *inserts*, *updates*, *selects* etc. Essas ações são passadas ao objeto `Database`, que encapsula o nosso pool de conexões, conexões estas que são utilizadas pelo `Database` para executar os comandos.

No caso de consultas que retornam múltiplos registros, vamos utilizar o método `stream`. Esse método permite que consumamos os registros na medida em que são lidos da base, sem que seja necessário carregar todos os dados na memória da aplicação antes que eles possam ser processados.

Agora que temos os conceitos básicos compreendidos, vamos criar o primeiro dos nossos três repositórios. No código a seguir, criamos a estrutura para a entidade produto da nossa aplicação, criamos o seu `TableQuery` e construímos um método para criar produtos:

```
package com.casadocodigo.repository

import com.casadocodigo.repository.DBConnection.db
import slick.basic.DatabasePublisher
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import scala.concurrent.Future

case class Produto(id: Long, descricao: String, preco:
Double)

class ProdutoSchema(tag: Tag) extends Table[Produto]
(tag, "produto") {
  def id = column[Long]("id", O.PrimaryKey)

  def descricao = column[String]("descricao")

  def preco = column[Double]("preco")
```

```
  def * = (id, descricao, preco) <> (Produto.tupled,
Produto.unapply)
}

object RepositorioDeProdutos extends DBConnection {
  private val tabela = TableQuery[ProdutoSchema]
  val schema = tabela.schema
  db.run(DBIO.seq(
    schema.createIfNotExists
  ))

  def criar(produto: Produto): Future[Produto] = run {
    (tabela returning tabela) += produto
  }

}
```

Após criarmos o nosso repositório, vamos fazer uma simples modificação de teste no nosso objeto `Boot`. Vamos mandar criar um novo produto para testar se toda a nossa configuração/codificação está correta:

```
package com.casadocodigo

import akka.actor.ActorSystem
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.http.scaladsl.server.Directives._
import com.casadocodigo.repository.{Produto,
RepositorioDeProdutos}

import scala.concurrent.ExecutionContextExecutor
import scala.io.StdIn
```

```
object Boot extends App {

  implicit val system: ActorSystem =
ActorSystem("system")
  implicit val executionContext:
ExecutionContextExecutor = system.dispatcher

  val route =
    path("hello") {
      get {

complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "
<h1>Olá, Akka!</h1>"))
      }
    }

  val bindingFuture = Http().newServerAt("0.0.0.0",
8080).bind(route)
  RepositorioDeProdutos.criar(Produto(1, "abc",
1)).onComplete(
    p =>
      println(p)
  )

  StdIn.readLine()
  bindingFuture
    .flatMap(_.unbind())
    .onComplete(_ => system.terminate())
}
```

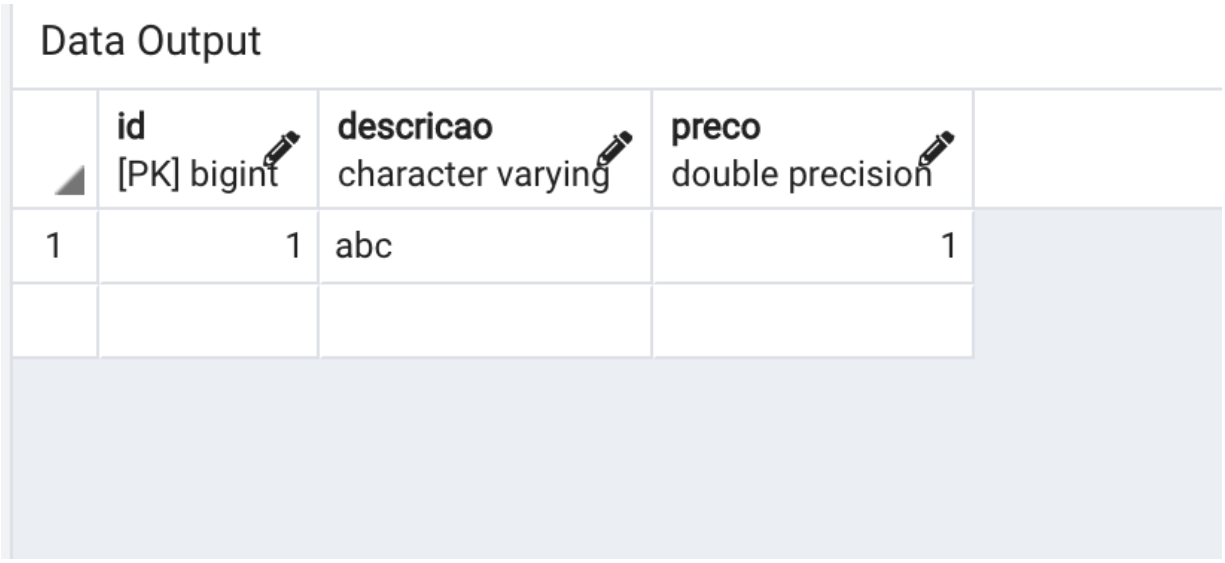
Vamos executar nosso código. Também aproveitaremos para testar as configurações que permitem executar a aplicação tanto via contêiner quanto via sbt.

Primeiramente, testaremos via contêiner. Para isso, basta executar o script `run_application.sh`. Como nosso script executa os contêineres em background, não vamos observar nenhuma diferença em relação às execuções anteriores.

Para verificar se tudo correu bem, vamos nos conectar ao banco de dados. Utilize seu programa favorito para acessar a bases de dados - eu utilizo o PG admin 4 - e conecte-se ao banco seguindo estas configurações:

- host: localhost
- port: 5432
- database: ecommerce
- user: postgres
- password: teste

Após nos conectarmos, navegamos até o *schema* public do banco e verificamos que a tabela não só foi criada com sucesso, como o nosso produto foi inserido com sucesso:



Data Output

| | id [PK] bigint | descricao character varying | preco double precision |
|---|---|---|---|
| 1 | 1 | abc | 1 |
| | | | |

Figura 5.5: Produto criado na base de dados.

Agora, faremos um teste executando a aplicação via sbt. Vamos destruir o nosso banco de dados e a API executando:

```
docker-compose down
```

Em seguida, vamos criar outro script, chamado `run_local.sh`. Esse script engloba os comandos necessários para subir o banco de dados e a aplicação via sbt:

```
#!/usr/bin/env bash

docker-compose up -d --scale api=0

sleep 10

sbt run
```

Utilizamos o comando `sleep` aqui para dar tempo ao banco de dados para terminar de inicializar antes de subirmos a API.

Vamos executar o script. Observaremos uma grande quantidade de logs sendo emitida, pois estamos rodando o Slick em modo debug por padrão. Os logs nos mostram diversas coisas, como as threads do pool de conexões da aplicação, como podemos ver no fragmento a seguir:

```
...
21:00:43.045 [database.postgres connection adder] DEBUG
com.zaxxer.hikari.pool.HikariPool - database.postgres -
Added connection
org.postgresql.jdbc.PgConnection@a192e6a
21:00:43.061 [database.postgres connection adder] DEBUG
com.zaxxer.hikari.pool.HikariPool - database.postgres -
Added connection
org.postgresql.jdbc.PgConnection@ac9fa87
21:00:43.075 [database.postgres connection adder] DEBUG
com.zaxxer.hikari.pool.HikariPool - database.postgres -
Added connection
org.postgresql.jdbc.PgConnection@748d49b8
21:01:12.863 [database.postgres housekeeper] DEBUG
com.zaxxer.hikari.pool.HikariPool - database.postgres -
Pool stats (total=10, active=0, idle=10, waiting=0)
21:01:42.866 [database.postgres housekeeper] DEBUG
com.zaxxer.hikari.pool.HikariPool - database.postgres -
Pool stats (total=10, active=0, idle=10, waiting=0)
...
```

Se nos reconectarmos ao banco, vamos ver que o produto foi criado com sucesso, provando que nossas configurações foram todas um sucesso! Agora, vamos apenas remover a linha `RepositorioDeProdutos.criar(Produto(1, "abc", 1)).onComplete(p => println(p))` do nosso objeto `Boot`, dado que não queremos que nossa aplicação tente criar esse produto toda vez que for iniciada.

Antes de prosseguirmos para os próximos repositórios, vamos implementar o restante dos métodos desse repositório que utilizaremos. Vamos implementar simples operações, que consistem em atualizar um produto, remover um produto, buscá-lo por seu id e buscar produtos pela descrição, em um estilo *LIKE*. Também vamos modificar a definição da chave primária da tabela, fazendo com que ela autoincremente o id nas inserções, deixando essa tarefa para o banco de dados:

```
package com.casadocodigo.repository

import com.casadocodigo.repository.DBConnection.db
import slick.basic.DatabasePublisher
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import scala.concurrent.Future
```

```
case class Produto(id: Long, descricao: String, preco:
Double)

class ProdutoSchema(tag: Tag) extends Table[Produto]
(tag, "produto") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)

  def descricao = column[String]("descricao")

  def preco = column[Double]("preco")

  def * = (id, descricao, preco) <> (Produto.tupled,
Produto.unapply)
}

object RepositorioDeProdutos extends DBConnection {
  private val tabela = TableQuery[ProdutoSchema]
  val schema = tabela.schema
  db.run(DBIO.seq(
    schema.createIfNotExists
  ))

  def criar(produto: Produto): Future[Produto] = run {
    (tabela returning tabela) += produto
  }

  def atualizar(produto: Produto): Future[Int] = run {
    tabela.filter(_.id === produto.id).update(produto)
  }

  def remover(produtoId: Long): Future[Int] = run {
    tabela.filter(_.id === produtoId).delete
  }

  def buscarPorId(produtoId: Long):
DatabasePublisher[Produto] = stream {
    tabela.filter(_.id === produtoId).result
```

```
  }

  def buscarPorDescricao(desc: String):
DatabasePublisher[Produto] = stream {
    tabela.filter(_.descricao like s"%$desc%").result
  }

}
```

Vamos criar os outros repositórios da aplicação. A seguir, criaremos o repositório da entidade cliente:

```
package com.casadocodigo.repository

import com.casadocodigo.Boot.executionContext
import com.casadocodigo.repository.DBConnection.db
import
com.casadocodigo.repository.RepositorioDeClientes.
{tabela, tabelaFilha, tabelaPedidos}
import slick.basic.DatabasePublisher
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import scala.concurrent.Future

case class Cliente(id: Long, nome: String) {
  def comEnderecos(): Future[Seq[Endereco]] = {
    DBConnection.db.run((for {
      (_, end) <- tabela filter (_.id === id) join
tabelaFilha on (_.id === _.clienteId)
    } yield end).result)
  }
}

class ClienteSchema(tag: Tag) extends Table[Cliente]
```

```
(tag, "cliente") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)

  def nome = column[String]("nome")

  def * = (id, nome) <> (Cliente.tupled,
Cliente.unapply)
}

case class Endereco(id: Long, rua: String, numero:
Long, cidade: String, estado: String, cep: String,
clienteId: Long)

class EnderecoSchema(tag: Tag) extends Table[Endereco]
(tag, "endereco") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)

  def rua = column[String]("rua")

  def numero = column[Long]("numero")

  def cidade = column[String]("cidade")

  def estado = column[String]("estado")

  def cep = column[String]("cep")

  def clienteId = column[Long]("cliente_id")

  def cliente = foreignKey("cliente", clienteId,
tabela)(_.id, onDelete = ForeignKeyAction.Cascade)


  def * = (id, rua, numero, cidade, estado, cep,
clienteId) <> (Endereco.tupled, Endereco.unapply)
}
```

```
object RepositorioDeClientes extends DBConnection {
  val tabelaPedidos = TableQuery[PedidoSchema]
  val tabelaFilha = TableQuery[EnderecoSchema]
  val tabela = TableQuery[ClienteSchema]
  db.run(DBIO.seq(
    tabela.schema.createIfNotExists,
    tabelaFilha.schema.createIfNotExists
  ))

  def criar(cliente: Cliente, enderecos:
List[Endereco]): Future[(Cliente, Option[Int])] = {
    val commands = for {
      cliente <- tabela returning tabela += cliente
      endrs <- tabelaFilha ++= enderecos.map(end => {
        end.copy(clienteId = cliente.id)
      })
    } yield (cliente, endrs)
    run(commands)
  }

  def atualizar(cliente: Cliente, enderecos:
List[Endereco]): Future[Int] = {
    enderecos.foreach(end => {
      run(tabelaFilha.filter(_.id ===
end.id).update(end))
    })
    run(tabela.filter(_.id ===
cliente.id).update(cliente))
  }

  def remover(clienteId: Long): Future[Int] = {
    run(tabela.filter(_.id === clienteId).delete)
    run(tabelaFilha.filter(_.clienteId ===
clienteId).delete)
  }

  def buscarPorId(clienteId: Long):
```

```
DatabasePublisher[Cliente] = stream {
    tabela.filter(_.id === clienteId).result
  }

}
```

Neste repositório, vemos nosso primeiro relacionamento entre entidades. Um ponto importante a se destacar do Slick é que, embora ele tenha algumas características em comum com frameworks ORM (*Object-Relational Mapping*), o Slick não possui um mapeamento 100% orientado a objetos quando mapeamos relacionamentos.

Por isso, mapeamos manualmente os relacionamentos, como podemos ver nos métodos de criação e atualização, que relacionamos associando os ids, e também criamos um método utilitário chamado `comEnderecos`, que realiza uma query estilo *join* entre as tabelas, que será muito útil para queries. Para exemplificar a utilidade, vamos supor que existem dois clientes com dois endereços cada, conforme as imagens:

| | id [PK] bigint | nome character varying |
|---|---|---|
| 1 | 1 | Alexandre |
| 2 | 2 | Alexandre 2 |
| | | |

Figura 5.6: Clientes criados na base de dados.

| | id [PK] bigint | rua character varying | numero bigint | cidade character varying | estado character varying | cep character varying | cliente_id bigint |
|---|---|---|---|---|---|---|---|
| 1 | 1 | rua teste | 123 | São Paulo | SP | 03423100 | 1 |
| 2 | 2 | rua teste 2 | 1234 | São Paulo | SP | 03423100 | 1 |
| 3 | 3 | rua teste 3 | 123 | São Paulo | SP | 03423100 | 2 |
| 4 | 4 | rua teste 4 | 1234 | São Paulo | SP | 03423100 | 2 |
| | | | | | | | |

Figura 5.7: Endereços criados na base de dados.

Dada essa estrutura, se quisermos imprimir os dados do cliente de id 2 e todos os seus endereços, basta escrevermos algo como isto:

```
Source.fromPublisher(RepositorioDeClientes.buscarPorId(
2))
    .runForeach(cli => {
      println(cli)
```

```
      cli.comEnderecos().foreach(
        end => {
          println(end)
        }
      )
    })
```

O `Source.fromPublisher` , que vemos nesse exemplo, é uma Akka Streams! Nos próximos capítulos, vamos aprender mais sobre Akka Streams, mas, neste momento, basta sabermos que esse código permite nos inscrevermos na stream e abrir um canal de consumo com a base de dados, consumindo todos os registros retornados pela consulta.

Por fim, vamos codificar o último dos nossos repositórios, o repositório de pedidos:

```
package com.casadocodigo.repository

import com.casadocodigo.Boot.executionContext
import com.casadocodigo.repository.DBConnection.db
import
com.casadocodigo.repository.RepositorioDePedidos.
{tabelaRelacionamentoProdutos, tabelaProdutos, tabela,
tabelaClientes}
import slick.basic.DatabasePublisher
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import scala.concurrent.Future
```

```
case class Pedido(id: Long, descricao: String,
clienteId: Long) {
  def comCliente(): Future[Seq[Cliente]] = {
    DBConnection.db.run((for {
      (_, cli) <- tabela filter (_.id === id) join
tabelaClientes on (_.clienteId === _.id)
    } yield cli).result)
  }

  def comProdutos(): Future[Seq[Produto]] = {
    DBConnection.db.run((for {
      (_, prod) <- (tabela filter (_.id === id) join
tabelaRelacionamentoProdutos on
        (_.id === _.pedidoId)) join tabelaProdutos on
(_._2.produtoId === _.id)
    } yield prod).result)
  }
}

case class PedidoProduto(pedidoId: Long, produtoId:
Long, quantidade: Long)

class PedidoProdutoSchema(tag: Tag) extends
Table[PedidoProduto](tag, "pedido_produto") {
  def pedidoId = column[Long]("pedido_id")

  def produtoId = column[Long]("produto_id")

  def quantidade = column[Long]("quantidade")

  def pedido = foreignKey("pedido_produto", pedidoId,
tabela)(_.id)

  def produto = foreignKey("produto_pedido", produtoId,
tabelaProdutos)(_.id, onDelete =
ForeignKeyAction.Cascade, onUpdate =
```

```
ForeignKeyAction.Cascade)

  def * = (pedidoId, produtoId, quantidade) <>
(PedidoProduto.tupled, PedidoProduto.unapply)
}

class PedidoSchema(tag: Tag) extends Table[Pedido](tag,
"pedido") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)

  def descricao = column[String]("descricao")

  def clienteId = column[Long]("cliente_id")

  def cliente = foreignKey("cliente_pedido", clienteId,
tabelaClientes)(_.id)

  def * = (id, descricao, clienteId) <> (Pedido.tupled,
Pedido.unapply)
}

object RepositorioDePedidos extends DBConnection {
  val tabelaRelacionamentoProdutos =
TableQuery[PedidoProdutoSchema]
  val tabelaProdutos = TableQuery[ProdutoSchema]
  val tabelaClientes = TableQuery[ClienteSchema]
  val tabela = TableQuery[PedidoSchema]
  db.run(DBIO.seq(
    tabela.schema.createIfNotExists,

tabelaRelacionamentoProdutos.schema.createIfNotExists
  ))

  def criar(pedido: Pedido, produtos:
List[PedidoProduto]): Future[(Pedido, Option[Int])] = {
    val commands = for {
      ped <- tabela returning tabela += pedido
```

```
      items <- tabelaRelacionamentoProdutos ++=
produtos.map(prd => {
        prd.copy(pedidoId = ped.id)
      })
    } yield (ped, items)
    run(commands)
  }

  def atualizar(pedido: Pedido): Future[Int] = {
    run(tabela.filter(_.id ===
pedido.id).update(pedido))
  }

  def remover(pedidoId: Long): Future[Int] = {
    run(tabela.filter(_.id === pedidoId).delete)
    run(tabelaRelacionamentoProdutos.filter(_.pedidoId
=== pedidoId).delete)
  }

  def buscarPorId(pedidoId: Long):
Future[DatabasePublisher[Pedido]] = Future {
    stream {
      tabela.filter(_.id === pedidoId).result
    }
  }
}
```

Neste repositório, voltamos a utilizar os conceitos de relacionamento entre tabelas, estabelecendo um relacionamento entre as tabelas pedido e cliente. Também criamos uma tabela de relacionamento entre pedido e produto, para estabelecer o relacionamento N x N entre essas tabelas. Também criamos alguns métodos utilitários que realizam os `join` entre as tabelas.

Assim, analogamente ao que fizemos anteriormente, se tivermos um pedido id 1:

| | id [PK] bigint | descricao character varying | cliente_id bigint |
|---|---|---|---|
| 1 | 1 | teste | 1 |

Figura 5.8: Pedido criado na base de dados.

Que por sua vez possui 1 item de quantidade 1:

| | pedido_id bigint | produto_id bigint | quantidade bigint |
|---|---|---|---|
| 1 | 1 | 1 | 2 |

Figura 5.9: Item do pedido criado na base de dados.

Podemos fazer uma consulta que retorne todos os dados de um pedido - seu cliente e seus produtos - escrevendo algo como isto:

```
Source.fromPublisher(RepositorioDePedidos.buscarPorId(1
))
    .runForeach(p => {
      println(p)
      p.comCliente().foreach(
        cli => println(cli)
      )
      p.comProdutos().foreach(
        prod => {
          println(prod)
        }
      )
    })
```

Concluindo a nossa codificação dos repositórios, modificaremos novamente o repositório de clientes. Vamos apenas incluir um método que permita que, a partir do cliente, seja possível consultar os seus pedidos:

```
...
case class Cliente(id: Long, nome: String) {
  def comEnderecos(): Future[Seq[Endereco]] = {
    DBConnection.db.run((for {
      (_, end) <- tabela filter (_.id === id) join
tabelaFilha on (_.id === _.clienteId)
    } yield end).result)
  }

  def comPedidos(): Future[Seq[Pedido]] = {
    DBConnection.db.run((for {
      (_, ped) <- tabela filter (_.id === id) join
tabelaPedidos on (_.id === _.clienteId)
    } yield ped).result)
  }
```

```
}
...
```

Assim, o código completo do repositório de clientes fica conforme podemos ver a seguir:

```
package com.casadocodigo.repository

import com.casadocodigo.Boot.executionContext
import com.casadocodigo.repository.DBConnection.db
import
com.casadocodigo.repository.RepositorioDeClientes.
{tabela, tabelaFilha, tabelaPedidos}
import slick.basic.DatabasePublisher
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import scala.concurrent.Future

case class Cliente(id: Long, nome: String) {
  def comEnderecos(): Future[Seq[Endereco]] = {
    DBConnection.db.run((for {
      (_, end) <- tabela filter (_.id === id) join
tabelaFilha on (_.id === _.clienteId)
    } yield end).result)
  }

  def comPedidos(): Future[Seq[Pedido]] = {
    DBConnection.db.run((for {
      (_, ped) <- tabela filter (_.id === id) join
tabelaPedidos on (_.id === _.clienteId)
    } yield ped).result)
  }
}
```

```
class ClienteSchema(tag: Tag) extends Table[Cliente]
(tag, "cliente") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)

  def nome = column[String]("nome")

  def * = (id, nome) <> (Cliente.tupled,
Cliente.unapply)
}

case class Endereco(id: Long, rua: String, numero:
Long, cidade: String, estado: String, cep: String,
clienteId: Long)

class EnderecoSchema(tag: Tag) extends Table[Endereco]
(tag, "endereco") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)

  def rua = column[String]("rua")

  def numero = column[Long]("numero")

  def cidade = column[String]("cidade")

  def estado = column[String]("estado")

  def cep = column[String]("cep")

  def clienteId = column[Long]("cliente_id")

  def cliente = foreignKey("cliente", clienteId,
tabela)(_.id, onDelete = ForeignKeyAction.Cascade)


  def * = (id, rua, numero, cidade, estado, cep,
clienteId) <> (Endereco.tupled, Endereco.unapply)
}
```

```
object RepositorioDeClientes extends DBConnection {
  val tabelaPedidos = TableQuery[PedidoSchema]
  val tabelaFilha = TableQuery[EnderecoSchema]
  val tabela = TableQuery[ClienteSchema]
  db.run(DBIO.seq(
    tabela.schema.createIfNotExists,
    tabelaFilha.schema.createIfNotExists
  ))

  def criar(cliente: Cliente, enderecos:
List[Endereco]): Future[(Cliente, Option[Int])] = {
    val commands = for {
      cliente <- tabela returning tabela += cliente
      endrs <- tabelaFilha ++= enderecos.map(end => {
        end.copy(clienteId = cliente.id)
      })
    } yield (cliente, endrs)
    run(commands)
  }

  def atualizar(cliente: Cliente, enderecos:
List[Endereco]): Future[Int] = {
    enderecos.foreach(end => {
      run(tabelaFilha.filter(_.id ===
end.id).update(end))
    })
    run(tabela.filter(_.id ===
cliente.id).update(cliente))
  }

  def remover(clienteId: Long): Future[Int] = {
    run(tabela.filter(_.id === clienteId).delete)
    run(tabelaFilha.filter(_.clienteId ===
clienteId).delete)
  }
```

```
  def buscarPorId(clienteId: Long):
DatabasePublisher[Cliente] = stream {
    tabela.filter(_.id === clienteId).result
  }

}
```

Agora que concluímos a camada de persistência, vamos começar a codificar o código que fará a junção entre a persistência e a comunicação, disponibilizando os endpoints que compõem a nossa API.

## 5.5 Criando os serviços da aplicação

Vamos codificar as classes de serviço. Essas classes serão responsáveis por invocar os repositórios que criamos e serão implementadas como atores, de modo a permitir que elas sejam invocadas de maneira assíncrona.

Vamos começar criando o nosso serviço de produtos, que encapsula os métodos de gerenciamento de produtos:

```
package com.casadocodigo.service

import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{ActorRef, Behavior,
SupervisorStrategy}
import akka.event.slf4j.Logger
```

```
import com.casadocodigo.Boot.executionContext
import com.casadocodigo.repository.{Produto,
RepositorioDeProdutos}
import slick.basic.DatabasePublisher

import scala.language.postfixOps
import scala.util.{Failure, Success}

object ServicoDeProdutos {

  val logger = Logger("ServicoDeProdutos")

  trait MensagemProduto

  trait RespostaProduto

  case class MensagemCriarProduto(produto: Produto,
replyTo: ActorRef[RespostaProduto]) extends
MensagemProduto

  case class MensagemAtualizarProduto(produto: Produto,
replyTo: ActorRef[RespostaProduto]) extends
MensagemProduto

  case class MensagemRemoverProduto(produtoId: Long,
replyTo: ActorRef[RespostaProduto]) extends
MensagemProduto

  case class MensagemBuscarProdutoPorId(id: Long,
replyTo: ActorRef[RespostaProduto]) extends
MensagemProduto

  case class
MensagemBuscarProdutoPorDescricao(descricao: String,
replyTo: ActorRef[RespostaProduto]) extends
MensagemProduto
```

```
  case class RespostaGerenciamentoDeProduto() extends
RespostaProduto

  case class RespostaGerenciamentoDeProdutoFalha()
extends RespostaProduto

  case class RespostaBuscaDeProduto(publicador:
DatabasePublisher[Produto]) extends RespostaProduto

  case class RespostaBuscaDeProdutoFalha() extends
RespostaProduto

  def apply(): Behavior[MensagemProduto] =
Behaviors.supervise[MensagemProduto](behavior())
    .onFailure[Exception](SupervisorStrategy.restart)

  def behavior(): Behavior[MensagemProduto] =
Behaviors.receive {
    (_, mensagem) =>
      mensagem match {
        case MensagemCriarProduto(produto, replyTo) =>

RepositorioDeProdutos.criar(produto).onComplete {
            case Success(_) => replyTo !
RespostaGerenciamentoDeProduto()
            case Failure(e) => logger.error(f"erro ao
tentar criar o produto: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDeProdutoFalha()
          }
          Behaviors.same
        case MensagemAtualizarProduto(produto, replyTo)
=>

RepositorioDeProdutos.atualizar(produto).onComplete {
            case Success(_) => replyTo !
RespostaGerenciamentoDeProduto()
```

```
            case Failure(e) => logger.error(f"erro ao
tentar atualizar o produto: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDeProdutoFalha()
          }
          Behaviors.same
        case MensagemRemoverProduto(id, replyTo) =>
          RepositorioDeProdutos.remover(id).onComplete
{
            case Success(_) => replyTo !
RespostaGerenciamentoDeProduto()
            case Failure(e) => logger.error(f"erro ao
tentar remover o produto: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDeProdutoFalha()
          }
          Behaviors.same
        case MensagemBuscarProdutoPorId(id, replyTo) =>

RepositorioDeProdutos.buscarPorId(id).onComplete {
            case Success(response) => replyTo !
RespostaBuscaDeProduto(response)
            case Failure(e) => logger.error(f"erro ao
tentar buscar o produto: ${e.getMessage}")
              replyTo ! RespostaBuscaDeProdutoFalha()
          }
          Behaviors.same
        case
MensagemBuscarProdutoPorDescricao(descricao, replyTo)
=>

RepositorioDeProdutos.buscarPorDescricao(descricao).onC
omplete {
            case Success(response) => replyTo !
RespostaBuscaDeProduto(response)
            case Failure(e) => logger.error(f"erro ao
tentar buscar produtos por descricao: ${e.getMessage}")
```

```
                replyTo ! RespostaBuscaDeProdutoFalha()
            }
            Behaviors.same
        }


    }

}
```

> Você pode ter reparado que não utilizamos o logger fornecido pelo contexto do ator. Devido ao logger não ser thread-safe, existem cenários onde não é possível utilizar esse logger, como dentro de uma Future. Por esse motivo, criamos manualmente o logger neste cenário.

Nosso serviço cria uma hierarquia de mensagens, onde todas as requisições para o ator são feitas através de uma mensagem do tipo `MensagemProduto` e todas as respostas são mensagens do tipo `RespostaProduto`. Isso nos permite criar um único ator para o consumo das mensagens quando implementarmos a camada que utilizará esse ator.

Nossas mensagens de requisição recebem como parâmetro uma referência para outro ator que espera receber uma resposta da operação. Esse detalhe nas nossas mensagens faz parte do padrão *Ask*

(responder), que compreenderemos melhor no próximo capítulo.

Todas as chamadas que realizamos para os repositórios são feitas em conjunto com Futures, que nos permitem desacoplar os atores de serviço das operações de banco.

Por conta disso, vamos precisar fazer uma pequena alteração nos repositórios que envolve os métodos de consultas ao banco que estão em objetos do tipo `Futures`, de forma que possamos utilizar o mesmo padrão para todas as operações. Desse modo, no repositório de produtos, modificamos os métodos assim:

```
...
def buscarPorId(produtoId: Long):
Future[DatabasePublisher[Produto]] = Future {
    stream {
      tabela.filter(_.id === produtoId).result
    }
  }

  def buscarPorDescricao(desc: String):
Future[DatabasePublisher[Produto]] = Future {
    stream {
      tabela.filter(_.descricao like s"%$desc%").result
    }
  }
 ...
```

E fazemos as mesmas mudanças nos repositórios de clientes e pedidos:

```
...
def buscarPorId(clienteId: Long):
Future[DatabasePublisher[Cliente]] = Future {
    stream {
      tabela.filter(_.id === clienteId).result
    }
  }
...
```

```
...
def buscarPorId(pedidoId: Long):
Future[DatabasePublisher[Pedido]] = Future {
    stream {
      tabela.filter(_.id === pedidoId).result
    }
  }
...
```

Agora que compreendemos como os serviços funcionam, vamos implementar os serviços restantes. Primeiramente, vamos criar o serviço de clientes:

```
package com.casadocodigo.service

import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{ActorRef, Behavior,
SupervisorStrategy}
import akka.event.slf4j.Logger
import com.casadocodigo.Boot.executionContext
import com.casadocodigo.repository.{Cliente, Endereco,
RepositorioDeClientes}
```

```
import slick.basic.DatabasePublisher

import scala.language.postfixOps
import scala.util.{Failure, Success}

object ServicoDeClientes {

  val logger = Logger("ServicoDeClientes")

  trait MensagemCliente

  trait RespostaCliente

  case class MensagemCriarCliente(cliente: Cliente,
enderecos: List[Endereco], replyTo:
ActorRef[RespostaCliente]) extends MensagemCliente

  case class MensagemAtualizarCliente(cliente: Cliente,
enderecos: List[Endereco], replyTo:
ActorRef[RespostaCliente]) extends MensagemCliente

  case class MensagemRemoverCliente(id: Long, replyTo:
ActorRef[RespostaCliente]) extends MensagemCliente

  case class MensagemBuscarClientePorId(id: Long,
replyTo: ActorRef[RespostaCliente]) extends
MensagemCliente


  case class RespostaGerenciamentoDeCliente() extends
RespostaCliente

  case class RespostaGerenciamentoDeClienteFalha()
extends RespostaCliente

  case class RespostaBuscaDeCliente(publicador:
DatabasePublisher[Cliente]) extends RespostaCliente
```

```
  case class RespostaBuscaDeClienteFalha() extends
RespostaCliente

  def apply(): Behavior[MensagemCliente] =
Behaviors.supervise[MensagemCliente](behavior())
    .onFailure[Exception](SupervisorStrategy.restart)

  def behavior(): Behavior[MensagemCliente] =
Behaviors.receive {
    (_, mensagem) =>
      mensagem match {
        case MensagemCriarCliente(cliente, enderecos,
replyTo) =>
          RepositorioDeClientes.criar(cliente,
enderecos).onComplete {
            case Success(_) => replyTo !
RespostaGerenciamentoDeCliente()
            case Failure(e) => logger.error(f"erro ao
tentar criar o cliente: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDeClienteFalha()
          }
          Behaviors.same
        case MensagemAtualizarCliente(cliente,
enderecos, replyTo) =>
          RepositorioDeClientes.atualizar(cliente,
enderecos).onComplete {
            case Success(_) => replyTo !
RespostaGerenciamentoDeCliente()
            case Failure(e) => logger.error(f"erro ao
tentar atualizar o cliente: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDeClienteFalha()
          }
          Behaviors.same
        case MensagemRemoverCliente(id, replyTo) =>
```

```
          RepositorioDeClientes.remover(id).onComplete
{
            case Success(_) => replyTo !
RespostaGerenciamentoDeCliente()
            case Failure(e) => logger.error(f"erro ao
tentar remover o cliente: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDeClienteFalha()
          }
          Behaviors.same
        case MensagemBuscarClientePorId(id, replyTo) =>

RepositorioDeClientes.buscarPorId(id).onComplete {
            case Success(response) => replyTo !
RespostaBuscaDeCliente(response)
            case Failure(e) => logger.error(f"erro ao
tentar buscar o cliente: ${e.getMessage}")
              replyTo ! RespostaBuscaDeClienteFalha()
          }
          Behaviors.same
      }

  }

}
```

Por fim, vamos criar o serviço de pedidos:

```
package com.casadocodigo.service

import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{ActorRef, Behavior,
SupervisorStrategy}
import akka.event.slf4j.Logger
import com.casadocodigo.Boot.executionContext
import com.casadocodigo.repository.{Pedido,
PedidoProduto, RepositorioDePedidos}
```

```
import slick.basic.DatabasePublisher

import scala.language.postfixOps
import scala.util.{Failure, Success}

object ServicoDePedidos {

  val logger = Logger("ServicoDePedidos")

  trait MensagemPedido

  trait RespostaPedido

  case class MensagemCriarPedido(pedido: Pedido,
produtos: List[PedidoProduto], replyTo:
ActorRef[RespostaPedido]) extends MensagemPedido

  case class MensagemAtualizarPedido(pedido: Pedido,
replyTo: ActorRef[RespostaPedido]) extends
MensagemPedido

  case class MensagemRemoverPedido(id: Long, replyTo:
ActorRef[RespostaPedido]) extends MensagemPedido

  case class MensagemBuscarPedidoPorId(id: Long,
replyTo: ActorRef[RespostaPedido]) extends
MensagemPedido


  case class RespostaGerenciamentoDePedido() extends
RespostaPedido

  case class RespostaGerenciamentoDePedidoFalha()
extends RespostaPedido

  case class RespostaBuscaDePedido(publicador:
DatabasePublisher[Pedido]) extends RespostaPedido
```

```
  case class RespostaBuscaDePedidoFalha() extends
RespostaPedido

  def apply(): Behavior[MensagemPedido] =
Behaviors.supervise[MensagemPedido](behavior())
    .onFailure[Exception](SupervisorStrategy.restart)

  def behavior(): Behavior[MensagemPedido] =
Behaviors.receive {
    (_, mensagem) =>
      mensagem match {
        case MensagemCriarPedido(pedido, produtos,
replyTo) =>
          RepositorioDePedidos.criar(pedido,
produtos).onComplete {
            case Success(_) => replyTo !
RespostaGerenciamentoDePedido()
            case Failure(e) => logger.error(f"erro ao
tentar criar o pedido: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDePedidoFalha()
          }
          Behaviors.same
        case MensagemAtualizarPedido(pedido, replyTo)
=>

RepositorioDePedidos.atualizar(pedido).onComplete {
            case Success(_) => replyTo !
RespostaGerenciamentoDePedido()
            case Failure(e) => logger.error(f"erro ao
tentar atualizar o pedido: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDePedidoFalha()
          }
          Behaviors.same
        case MensagemRemoverPedido(id, replyTo) =>
```

```
          RepositorioDePedidos.remover(id).onComplete {
            case Success(_) => replyTo !
RespostaGerenciamentoDePedido()
            case Failure(e) => logger.error(f"erro ao
tentar remover o pedido: ${e.getMessage}")
              replyTo !
RespostaGerenciamentoDePedidoFalha()
          }
          Behaviors.same
        case MensagemBuscarPedidoPorId(id, replyTo) =>

RepositorioDePedidos.buscarPorId(id).onComplete {
            case Success(response) => replyTo !
RespostaBuscaDePedido(response)
            case Failure(e) => logger.error(f"erro ao
tentar buscar o pedido: ${e.getMessage}")
              replyTo ! RespostaBuscaDePedidoFalha()
          }
          Behaviors.same
      }

  }

}
```

Por ora, vamos seguir com a codificação e não faremos testes em separado dos serviços, pois deixaremos esta ação para o próximo capítulo, quando executarmos a API através dos seus endpoints. Vamos seguir agora para a última camada da nossa API, a camada de transporte.

CAPÍTULO 6

# Construindo uma API: parte 2 - criando os endpoints e testando

Continuando a construção da nossa API, vamos agora criar os endpoints, responsáveis por expor a aplicação para o mundo exterior. Também vamos utilizar o *Gatling*, uma biblioteca de testes de carga, a fim de avaliar como a nossa API lida sob um tráfego intenso de requisições.

## 6.1 Criando os endpoints da aplicação

Vamos criar a última camada de nossa aplicação, a camada de transporte. Ela será responsável por utilizar diretamente o Akka HTTP, criando as rotas HTTP e manipulando dados através da serialização provida pela biblioteca spray-json.

Vamos começar justamente por essa parte da codificação, a criação das configurações de serialização. Para isso, vamos criar um objeto chamado `Respostas` com o seguinte conteúdo:

```
package com.casadocodigo.route

import com.casadocodigo.repository.Produto
```

```
object Respostas {

  case class RespostaSucesso(sucesso: Boolean)

  case class RespostaBuscaProdutoSucesso(produto:
List[Produto])

}
```

Essas classes consistem nas respostas que vamos retornar nos endpoints de produtos: `RespostaSucesso` para as operações de criação, atualização e remoção de produtos e a classe `RespostaBuscaProdutoSucesso` para retornar uma lista de produtos para as operações de busca de produtos. Para as requisições dos endpoints, utilizaremos a própria entidade `Produto`, bem como classes primitivas como textos e números.

A seguir, vamos criar uma `trait` chamada `SerializadorJSON`. Essa trait estenderá as classes de suporte do spray-json e declarará implícitos, que serão responsáveis por serializar/deserializar as requisições e respostas dos endpoints:

```
package com.casadocodigo.route

import
akka.http.scaladsl.marshallers.sprayjson.SprayJsonSuppo
rt
import com.casadocodigo.repository.Produto
```

```
import com.casadocodigo.route.Respostas.
{RespostaBuscaProdutoSucesso, RespostaSucesso}
import spray.json.DefaultJsonProtocol

trait SerializadorJSON extends SprayJsonSupport with
DefaultJsonProtocol {

  implicit val respostaProdutoFormat =
jsonFormat3(Produto)
  implicit val respostaSucessoFormat =
jsonFormat1(RespostaSucesso)
  implicit val respostaProdutoBuscarPorIdFormat =
jsonFormat1(RespostaBuscaProdutoSucesso)

}
```

Vamos agora criar o arquivo de rotas de produtos, chamado `RotasDeProdutos`. Esse arquivo vai conter uma trait de mesmo nome do arquivo, estendendo a trait `SerializadorJSON`, que criamos anteriormente. Essa trait implementa cada rota através de um método diferente, juntando todas as definições dentro do método `rotasDeProdutos()`. Vamos observar todo o código agora e comentá-lo a seguir, para compreender melhor certos detalhes:

```
package com.casadocodigo.route

import akka.actor.typed.scaladsl.AskPattern.Askable
import akka.http.scaladsl.model.StatusCodes.BadRequest
import akka.http.scaladsl.server.Directives.{as,
complete, entity, failWith, onComplete, patch, path,
post, _}
```

```
import akka.http.scaladsl.server.Route
import akka.stream.scaladsl.{Sink, Source}
import com.casadocodigo.Boot.{atorDeProdutos,
executionContext, scheduler, system, timeout}
import com.casadocodigo.repository.Produto
import com.casadocodigo.route.Respostas.
{RespostaBuscaProdutoSucesso, RespostaSucesso}
import com.casadocodigo.service.ServicoDeProdutos
import com.casadocodigo.service.ServicoDeProdutos.
{MensagemAtualizarProduto,
MensagemBuscarProdutoPorDescricao,
MensagemBuscarProdutoPorId, MensagemCriarProduto,
MensagemRemoverProduto, RespostaGerenciamentoDeProduto}

import scala.concurrent.Future
import scala.language.postfixOps
import scala.util.{Failure, Success}

trait RotasDeProdutos extends SerializadorJSON {

  def rotasDeProdutos(): Route = criarProduto() ~
atualizarProduto() ~ removerProduto() ~
buscarPorIdProduto() ~ buscarPorDescricaoProduto()

  def criarProduto(): Route = post {
    path("produto") {
      entity(as[Produto]) { prod =>

        val response:
Future[ServicoDeProdutos.RespostaProduto] =
atorDeProdutos.ask(ref => MensagemCriarProduto(prod,
ref))
        onComplete(response) {
          case Success(response) => response match {
            case RespostaGerenciamentoDeProduto() =>
complete(RespostaSucesso(true))
            case _ => complete(BadRequest)
```

```
          }
          case Failure(e) => failWith(e)
        }
      }
    }
  }

  def atualizarProduto(): Route = patch {
    path("produto") {
      entity(as[Produto]) { prod =>

        val response:
Future[ServicoDeProdutos.RespostaProduto] =
atorDeProdutos.ask(ref =>
MensagemAtualizarProduto(prod, ref))
        onComplete(response) {
          case Success(response) => response match {
            case RespostaGerenciamentoDeProduto() =>
complete(RespostaSucesso(true))
            case _ => complete(BadRequest)
          }
          case Failure(e) => failWith(e)
        }
      }
    }
  }

  def removerProduto(): Route = delete {
    path("produto" / Segment) { id =>
      val response:
Future[ServicoDeProdutos.RespostaProduto] =
atorDeProdutos.ask(ref =>
MensagemRemoverProduto(id.toLong, ref))
      onComplete(response) {
        case Success(response) => response match {
          case RespostaGerenciamentoDeProduto() =>
complete(RespostaSucesso(true))
```

```
          case _ => complete(BadRequest)
        }
        case Failure(e) => failWith(e)
      }
    }
  }

  def buscarPorIdProduto(): Route = get {
    path("produto" / "id" / Segment) { id =>
      val response:
Future[ServicoDeProdutos.RespostaProduto] =
atorDeProdutos.ask(ref =>
MensagemBuscarProdutoPorId(id.toLong, ref))
      onComplete(response) {
        case Success(response) => response match {
          case
ServicoDeProdutos.RespostaBuscaDeProduto(publicador) =>
            val data = Source.fromPublisher(publicador)
              .runWith(Sink.collection[Produto,
List[Produto]])
              .map {
                listaDeprodutos =>

RespostaBuscaProdutoSucesso(listaDeprodutos)
              }
            complete(data)
          case _ => complete(BadRequest)
        }

        case Failure(e) => failWith(e)
      }
    }
  }

  def buscarPorDescricaoProduto(): Route = get {
    path("produto" / "descricao" / Segment) { descricao
=>
```

```
      val response:
Future[ServicoDeProdutos.RespostaProduto] =
atorDeProdutos.ask(ref =>
MensagemBuscarProdutoPorDescricao(descricao, ref))
      onComplete(response) {
        case Success(response) => response match {
          case
ServicoDeProdutos.RespostaBuscaDeProduto(publicador) =>
            val data = Source.fromPublisher(publicador)
              .runWith(Sink.collection[Produto,
List[Produto]])
              .map {
                listaDeProdutos =>

RespostaBuscaProdutoSucesso(listaDeProdutos)
              }
            complete(data)
          case _ => complete(BadRequest)
        }

        case Failure(e) => failWith(e)
      }
    }
  }

}
```

Primeiro compreenderemos o que significam aqueles estranhos `~` no meio do código. Esses operadores especiais são definidos pelo Akka HTTP e implementam concatenações de rotas. Eles nos permitem juntarmos todas as rotas definidas no arquivo em um único ponto, a fim de facilitar a sua importação posteriormente.

Rotas no Akka HTTP têm suas implementações definidas a partir dos verbos HTTP. Assim, se quisermos implementar uma chamada GET, incluímos uma função chamada `get {....}` , `post {....}` para chamadas POST e assim por diante. Para definir os caminhos dos endpoints, utilizamos outra função chamada `path("path") {....}` . Essa função também pode ser configurada com um objeto especial chamado `Segment` , que permite definirmos *path parameters*, ou seja, parâmetros que passamos como parte da URL do endpoint.

No caso de endpoints que recebem objetos como entrada, como a entidade `Produto` , utilizamos outra função especial do spray, a `entity(as[Produto]) {....}` , dentro da qual recebemos como parâmetro o objeto `Produto` já devidamente deserializado e pronto para uso. O restante do código é bastante simples e direto: consiste de chamadas ao ator de serviço e retorno dos resultados.

Conforme já vimos anteriormente, toda a comunicação entre atores é feita de forma assíncrona, ou seja, só é possível enviar mensagens de um ator para o outro em sentido único, sem a possibilidade de se esperar por mensagens de retorno. Para cenários em que precisamos obter

uma resposta da nossa requisição, como nas consultas de produtos, por exemplo, utilizamos o padrão `Ask`. Nesse padrão, dois atores comunicam entre si. O ator chamador envia em suas mensagens um ator anônimo, que atua como um canal, a partir do qual espera receber uma resposta do ator chamado. Como toda a comunicação é feita de forma assíncrona, conseguimos obter assim um modelo próximo de uma *request-response*, sem com isso perder as características de IO não blocante. O diagrama a seguir ilustra essa comunicação:

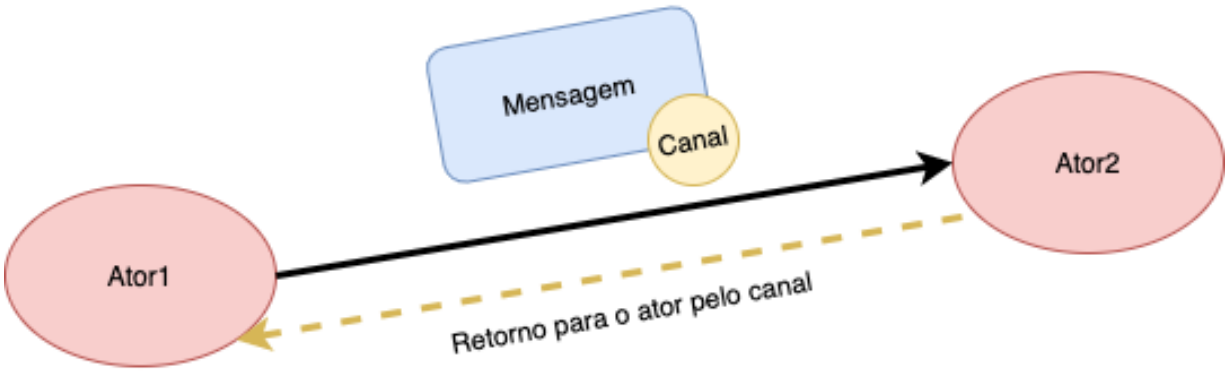



Figura 6.1: Padrão de comunicação ask.

Toda essa comunicação assíncrona é feita a partir de controles de monitoração, segundo os quais, caso uma resposta demore para ser retornada, a API aborta a requisição e um erro 500 é retornado para o cliente da API. Isso faz com que a aplicação seja resiliente, sem o risco de "prender" as requisições,

como vimos ocorrendo no capítulo 1. Mais à frente no capítulo, observaremos como esse controle é automaticamente gerenciado para nós pelo Akka.

Falando em controle de tempo de requisições, vamos fazer algumas refatorações na nossa aplicação. É possível notar no código que criamos uma configuração global de timeout para as camadas da aplicação, como vimos em trechos de código como `config.getInt("timeout")` , por exemplo. Vamos modificar agora o arquivo `application.base.conf` adicionando essa nova configuração:

```
timeout = 5

# Datasources
database.postgres {
  connectionPool = "HikariCP"
  dataSourceClass =
"org.postgresql.ds.PGSimpleDataSource"
  properties = {
    serverName = "localhost"
    serverName = ${?DB_HOST}
    portNumber = "5432"
    databaseName = "ecommerce"
    user = "postgres"
    password = "teste"
    password = ${?DB_PASS}
  }
  numThreads = 10
}
```

E vamos modificar o arquivo principal da aplicação, o `Boot`. Nesse arquivo, vamos definir diversos implícitos que são importados nos outros arquivos, como o sistema de atores e o contexto de execução. Também vamos concentrar no `Boot` a criação dos atores utilizados nas rotas criadas pelas traits, bem como importar todas as rotas criadas nas traits, através da herança múltipla do Scala, que nos permite que uma classe ou objeto estenda várias traits. Assim, esta é a nossa nova versão do objeto `Boot`:

```
package com.casadocodigo


import akka.actor.typed.Scheduler
import akka.actor.{ActorSystem, typed}
import akka.http.scaladsl.Http
import akka.actor.typed.scaladsl.adapter._
import akka.util.Timeout
import com.casadocodigo.route.RotasDeProdutos
import com.casadocodigo.service.ServicoDeProdutos
import com.typesafe.config.{Config, ConfigFactory}

import scala.concurrent.ExecutionContextExecutor
import scala.concurrent.duration.DurationInt
import scala.io.StdIn


object Boot extends App with RotasDeProdutos {

  implicit val config: Config =
ConfigFactory.load(Option(
    System.getenv("ENVIRONMENT"))
```

```
.getOrElse(Option(System.getProperty("ENVIRONMENT"))
      .getOrElse("application")))
  implicit val system: ActorSystem =
akka.actor.ActorSystem("ClassicToTypedSystem")
  implicit val executionContext:
ExecutionContextExecutor = system.dispatcher
  implicit val timeout: Timeout =
config.getInt("timeout").seconds
  val typedSystem: typed.ActorSystem[_] =
system.toTyped
  implicit val scheduler: Scheduler =
typedSystem.scheduler
  val atorDeProdutos =
typedSystem.systemActorOf(ServicoDeProdutos(),
"ServicoDeProdutos")

  val route = rotasDeProdutos()

  val bindingFuture = Http().newServerAt("0.0.0.0",
8080).bind(route)

  StdIn.readLine()
  bindingFuture
    .flatMap(_.unbind())
    .onComplete(_ => system.terminate())
}
```

Agora, vamos construir as nossas rotas de pedidos. Vamos criar a trait `RotasDePedidos` :

```
package com.casadocodigo.route

import akka.actor.typed.scaladsl.AskPattern.Askable
import akka.http.scaladsl.model.StatusCodes.BadRequest
import akka.http.scaladsl.server.Directives.{as,
```

```
complete, entity, failWith, onComplete, patch, path,
post, _}
import akka.http.scaladsl.server.Route
import akka.stream.scaladsl.{Sink, Source}
import com.casadocodigo.Boot.{atorDeEstoque,
atorDePedidos, config, executionContext, scheduler,
system, timeout}
import com.casadocodigo.repository.{Pedido,
PedidoProduto}
import
com.casadocodigo.route.Requisicoes.RequisicaoPedido
import com.casadocodigo.route.Respostas.
{RespostaBuscaPedidoSucesso, RespostaSucesso}
import com.casadocodigo.service.ServicoDeEstoque.
{ConsultarEstoque, RespostaConsultaEstoque}
import com.casadocodigo.service.ServicoDePedidos
import com.casadocodigo.service.ServicoDePedidos._

import scala.concurrent.duration.DurationInt
import scala.concurrent.{Await, Future}
import scala.language.postfixOps
import scala.util.{Failure, Success, Try}

trait RotasDePedidos extends SerializadorJSON {

  def rotasDePedidos(): Route = criarPedido() ~
atualizarPedido() ~ removerPedido() ~
buscarPorIdPedido()

  private def lift[T](futures: Seq[Future[T]]):
Seq[Future[Try[T]]] =
    futures.map(_.map {
      Success(_)
    }.recover { case t => Failure(t) })

  def waitAll[T](futures: Seq[Future[T]]):
Future[Seq[Try[T]]] =
```

```
    Future.sequence(lift(futures))

  def criarPedido(): Route = post {
    path("pedido") {
      entity(as[RequisicaoPedido]) { ped =>
        val hasOutOfStock =
Await.result(waitAll(ped.produtos.map(item =>
          atorDeEstoque.ask(ref =>
            ConsultarEstoque(item.produtoId, ref)
          )
        )), config.getInt("timeout") seconds).map(
          msg =>
            msg.getOrElse(RespostaConsultaEstoque(0,
disponivel = true))
        ).filter(msg =>
          msg match {
            case estoque: RespostaConsultaEstoque =>
              !estoque.disponivel
            case _ =>
              true
          }
        )

        if (hasOutOfStock.nonEmpty) {
          gerarRespostaDeErroDeEstoque()
        } else {
          gerarPedido(ped)
        }
      }
    }
  }

  private def gerarRespostaDeErroDeEstoque(): Route = {
    complete(BadRequest)
  }

  private def gerarPedido(ped: RequisicaoPedido): Route
```

```
= {
    val response:
Future[ServicoDePedidos.RespostaPedido] =
atorDePedidos.ask(ref =>
      MensagemCriarPedido(Pedido(0,
ped.pedido.descricao, ped.pedido.clienteId),
        ped.produtos.map({ item =>
          PedidoProduto(0, item.produtoId,
item.quantidade)
        }), ref))
    onComplete(response) {
      case Success(response) => response match {
        case RespostaGerenciamentoDePedido() =>
complete(RespostaSucesso(true))
        case _ => complete(BadRequest)
      }
      case Failure(e) => failWith(e)
    }
  }

  def atualizarPedido(): Route = patch {
    path("pedido") {
      entity(as[Pedido]) { ped =>

        val response:
Future[ServicoDePedidos.RespostaPedido] =
atorDePedidos.ask(ref => MensagemAtualizarPedido(ped,
ref))
        onComplete(response) {
          case Success(response) => response match {
            case RespostaGerenciamentoDePedido() =>
complete(RespostaSucesso(true))
            case _ => complete(BadRequest)
          }
          case Failure(e) => failWith(e)
        }
      }
```

```
    }
  }

  def removerPedido(): Route = delete {
    path("pedido" / Segment) { id =>
      val response:
Future[ServicoDePedidos.RespostaPedido] =
atorDePedidos.ask(ref =>
MensagemRemoverPedido(id.toLong, ref))
      onComplete(response) {
        case Success(response) => response match {
          case RespostaGerenciamentoDePedido() =>
complete(RespostaSucesso(true))
          case _ => complete(BadRequest)
        }
        case Failure(e) => failWith(e)
      }
    }
  }

  def buscarPorIdPedido(): Route = get {
    path("pedido" / "id" / Segment) { id =>
      val response:
Future[ServicoDePedidos.RespostaPedido] =
atorDePedidos.ask(ref =>
MensagemBuscarPedidoPorId(id.toLong, ref))
      onComplete(response) {
        case Success(response) => response match {
          case
ServicoDePedidos.RespostaBuscaDePedido(publicador) =>
            val data = Source.fromPublisher(publicador)
              .runWith(Sink.collection[Pedido,
List[Pedido]])
              .map {
                listaDePedidos =>

RespostaBuscaPedidoSucesso(listaDePedidos.head)
```

```
              }
            complete(data)
          case _ => complete(BadRequest)
        }

        case Failure(e) => failWith(e)
      }
    }
  }

}
```

Uma novidade nessas rotas é que também construímos classes de requisição. Isso foi feito de modo que, na criação dos pedidos, nossa API não exija que forneçamos o id do pedido que será criado em todos os objetos JSON da requisição, o que seria estranho.

Para criar essas classes de requisição, criamos o objeto `Requisicoes` e as classes:

```
package com.casadocodigo.route


object Requisicoes {

  case class RequisicaoPedidoDetalhe(descricao: String,
clienteId: Long)
  case class RequisicaoPedidoItem(produtoId: Long,
quantidade: Long)
  case class RequisicaoPedido(pedido:
RequisicaoPedidoDetalhe, produtos:
List[RequisicaoPedidoItem])
```

```
}
```

Essas classes também precisam ser serializadas. Vamos atualizar nossa trait de serialização:

```
package com.casadocodigo.route

import
akka.http.scaladsl.marshallers.sprayjson.SprayJsonSuppo
rt
import com.casadocodigo.repository.{Pedido,
PedidoProduto, Produto}
import com.casadocodigo.route.Requisicoes.
{RequisicaoPedido, RequisicaoPedidoDetalhe,
RequisicaoPedidoItem}
import com.casadocodigo.route.Respostas.
{RespostaBuscaPedidoSucesso,
RespostaBuscaProdutoSucesso, RespostaSucesso}
import spray.json.DefaultJsonProtocol

trait SerializadorJSON extends SprayJsonSupport with
DefaultJsonProtocol {

  implicit val respostaProdutoFormat =
jsonFormat3(Produto)
  implicit val respostaPedidoFormat =
jsonFormat3(Pedido)
  implicit val respostaPedidoProdutoFormat =
jsonFormat3(PedidoProduto)
  implicit val requisicaoPedidoDetalheFormat =
jsonFormat2(RequisicaoPedidoDetalhe)
  implicit val requisicaoPedidoItemFormat =
jsonFormat2(RequisicaoPedidoItem)
  implicit val requisicaoPedidoFormat =
jsonFormat2(RequisicaoPedido)
```

```
  implicit val respostaSucessoFormat =
jsonFormat1(RespostaSucesso)
  implicit val respostaProdutoBuscarPorIdFormat =
jsonFormat1(RespostaBuscaProdutoSucesso)
  implicit val respostaPedidoBuscarPorIdFormat =
jsonFormat1(RespostaBuscaPedidoSucesso)

}
```

Antes de voltarmos ao objeto `Boot` para atualizá-lo, criaremos também as nossas rotas de clientes. Essas rotas são bastante parecidas com o que acabamos de ver nas rotas de pedidos, então não necessitam de maiores explicações:

```
package com.casadocodigo.route

import akka.actor.typed.scaladsl.AskPattern.Askable
import akka.http.scaladsl.model.StatusCodes.BadRequest
import akka.http.scaladsl.server.Directives.{as,
complete, entity, failWith, onComplete, patch, path,
post, _}
import akka.http.scaladsl.server.Route
import akka.stream.scaladsl.{Sink, Source}
import com.casadocodigo.Boot.{atorDeClientes,
executionContext, scheduler, system, timeout}
import com.casadocodigo.repository.{Cliente, Endereco}
import
com.casadocodigo.route.Requisicoes.RequisicaoCliente
import com.casadocodigo.route.Respostas.
{RespostaBuscaClienteSucesso, RespostaSucesso}
import com.casadocodigo.service.ServicoDeClientes
import com.casadocodigo.service.ServicoDeClientes._

import scala.concurrent.Future
```

```
import scala.language.postfixOps
import scala.util.{Failure, Success}

trait RotasDeClientes extends SerializadorJSON {

  def rotasDeClientes(): Route = criarCliente() ~
atualizarCliente() ~ removerCliente() ~
buscarPorIdCliente()

  def criarCliente(): Route = post {
    path("cliente") {
      entity(as[RequisicaoCliente]) { cli =>

        val response:
Future[ServicoDeClientes.RespostaCliente] =
atorDeClientes.ask(ref =>
          MensagemCriarCliente(Cliente(0,
cli.detalhe.nome),
            cli.enderecos.map(end =>
              Endereco(0, end.rua, end.numero,
end.cidade,
                end.estado, end.cep, 0)), ref))
        onComplete(response) {
          case Success(response) => response match {
            case RespostaGerenciamentoDeCliente() =>
complete(RespostaSucesso(true))
            case _ => complete(BadRequest)
          }
          case Failure(e) => failWith(e)
        }
      }
    }
  }

  def atualizarCliente(): Route = patch {
    path("cliente") {
      entity(as[RequisicaoCliente]) { cli =>
```

```
        val response:
Future[ServicoDeClientes.RespostaCliente] =
atorDeClientes.ask(ref => MensagemAtualizarCliente(
          Cliente(cli.detalhe.id, cli.detalhe.nome),
          cli.enderecos.map(end =>
            Endereco(end.id, end.rua, end.numero,
end.cidade,
              end.estado, end.cep, cli.detalhe.id)),
ref))
        onComplete(response) {
          case Success(response) => response match {
            case RespostaGerenciamentoDeCliente() =>
complete(RespostaSucesso(true))
            case _ => complete(BadRequest)
          }
          case Failure(e) => failWith(e)
        }
      }
    }
  }

  def removerCliente(): Route = delete {
    path("cliente" / Segment) { id =>
      val response:
Future[ServicoDeClientes.RespostaCliente] =
atorDeClientes.ask(ref =>
MensagemRemoverCliente(id.toLong, ref))
      onComplete(response) {
        case Success(response) => response match {
          case RespostaGerenciamentoDeCliente() =>
complete(RespostaSucesso(true))
          case _ => complete(BadRequest)
        }
        case Failure(e) => failWith(e)
      }
    }
```

```
  }

  def buscarPorIdCliente(): Route = get {
    path("cliente" / "id" / Segment) { id =>
      val response:
Future[ServicoDeClientes.RespostaCliente] =
atorDeClientes.ask(ref =>
MensagemBuscarClientePorId(id.toLong, ref))
      onComplete(response) {
        case Success(response) => response match {
          case
ServicoDeClientes.RespostaBuscaDeCliente(publicador) =>
            val data = Source.fromPublisher(publicador)
              .runWith(Sink.collection[Cliente,
List[Cliente]])
              .map {
                listaDeClientes =>

RespostaBuscaClienteSucesso(listaDeClientes.head)
              }
            complete(data)
          case _ => complete(BadRequest)
        }

        case Failure(e) => failWith(e)
      }
    }
  }
}
```

Finalizando, vamos atualizar as classes de requisição, serializadores e o objeto `Boot` para criar os atores e instanciar as rotas. Estas são as alterações nas classes de requisições:

```
package com.casadocodigo.route


object Requisicoes {

  case class RequisicaoPedidoDetalhe(descricao: String,
clienteId: Long)
  case class RequisicaoPedidoItem(produtoId: Long,
quantidade: Long)
  case class RequisicaoPedido(pedido:
RequisicaoPedidoDetalhe, produtos:
List[RequisicaoPedidoItem])
  case class RequisicaoClienteDetalhe(id:Long, nome:
String)
  case class RequisicaoClienteEndereco(id:Long, rua:
String, numero: Long, cidade: String, estado: String,
cep: String)
  case class RequisicaoCliente(detalhe:
RequisicaoClienteDetalhe, enderecos:
List[RequisicaoClienteEndereco])

}
```

## As alterações nos serializadores:

```
package com.casadocodigo.route

import
akka.http.scaladsl.marshallers.sprayjson.SprayJsonSuppo
rt
import com.casadocodigo.repository.{Cliente, Endereco,
Pedido, PedidoProduto, Produto}
import com.casadocodigo.route.Requisicoes.
{RequisicaoCliente, RequisicaoClienteDetalhe,
RequisicaoClienteEndereco, RequisicaoPedido,
RequisicaoPedidoDetalhe, RequisicaoPedidoItem}
import com.casadocodigo.route.Respostas.
```

```
{RespostaBuscaClienteSucesso,
RespostaBuscaPedidoSucesso,
RespostaBuscaProdutoSucesso, RespostaSucesso}
import spray.json.DefaultJsonProtocol

trait SerializadorJSON extends SprayJsonSupport with
DefaultJsonProtocol {

  implicit val respostaProdutoFormat =
jsonFormat3(Produto)
  implicit val respostaPedidoFormat =
jsonFormat3(Pedido)
  implicit val respostaClienteFormat =
jsonFormat2(Cliente)
  implicit val respostaEnderecoFormat =
jsonFormat7(Endereco)
  implicit val respostaPedidoProdutoFormat =
jsonFormat3(PedidoProduto)
  implicit val requisicaoPedidoDetalheFormat =
jsonFormat2(RequisicaoPedidoDetalhe)
  implicit val requisicaoPedidoItemFormat =
jsonFormat2(RequisicaoPedidoItem)
  implicit val requisicaoPedidoFormat =
jsonFormat2(RequisicaoPedido)
  implicit val requisicaoClienteDetalheFormat =
jsonFormat2(RequisicaoClienteDetalhe)
  implicit val requisicaoClienteEnderecoFormat =
jsonFormat6(RequisicaoClienteEndereco)
  implicit val requisicaoClienteFormat =
jsonFormat2(RequisicaoCliente)
  implicit val respostaSucessoFormat =
jsonFormat1(RespostaSucesso)
  implicit val respostaProdutoBuscarPorIdFormat =
jsonFormat1(RespostaBuscaProdutoSucesso)
  implicit val respostaPedidoBuscarPorIdFormat =
jsonFormat1(RespostaBuscaPedidoSucesso)
  implicit val respostaClienteBuscarPorIdFormat =
```

```
jsonFormat1(RespostaBuscaClienteSucesso)


}
```

E, por fim, as alterações no objeto `Boot`:

```
package com.casadocodigo


import akka.actor.typed.Scheduler
import akka.actor.{ActorSystem, typed}
import akka.http.scaladsl.Http
import akka.actor.typed.scaladsl.adapter._
import akka.http.scaladsl.server.Directives._
import akka.util.Timeout
import com.casadocodigo.route.{RotasDeClientes,
RotasDePedidos, RotasDeProdutos}
import com.casadocodigo.service.{ServicoDeClientes,
ServicoDePedidos, ServicoDeProdutos}
import com.typesafe.config.{Config, ConfigFactory}

import scala.concurrent.ExecutionContextExecutor
import scala.concurrent.duration.DurationInt
import scala.io.StdIn


object Boot extends App with RotasDeProdutos with
RotasDePedidos with RotasDeClientes {

  implicit val config: Config =
ConfigFactory.load(Option(
    System.getenv("ENVIRONMENT"))

.getOrElse(Option(System.getProperty("ENVIRONMENT"))
      .getOrElse("application")))
  implicit val system: ActorSystem =
```

```
akka.actor.ActorSystem("ClassicToTypedSystem")
  implicit val executionContext:
ExecutionContextExecutor = system.dispatcher
  implicit val timeout: Timeout =
config.getInt("timeout").seconds
  val typedSystem: typed.ActorSystem[_] =
system.toTyped
  implicit val scheduler: Scheduler =
typedSystem.scheduler
  val atorDeProdutos =
typedSystem.systemActorOf(ServicoDeProdutos(),
"ServicoDeProdutos")
  val atorDePedidos =
typedSystem.systemActorOf(ServicoDePedidos(),
"ServicoDePedidos")
  val atorDeClientes =
typedSystem.systemActorOf(ServicoDeClientes(),
"ServicoDeClientes")

  val route = rotasDeProdutos() ~ rotasDePedidos() ~
rotasDeClientes()

  val bindingFuture = Http().newServerAt("0.0.0.0",
8080).bind(route)

  StdIn.readLine()
  bindingFuture
    .flatMap(_.unbind())
    .onComplete(_ => system.terminate())
}
```

E assim concluímos a construção da nossa API! Ainda falta incluirmos as chamadas à API externa - faremos isso em breve - mas, por ora, vamos fazer testes realizando chamadas ao que já construímos.

## 6.2 Testando os endpoints

Vamos começar a "brincar" com a nossa API, simulando chamadas de um uso real da aplicação. Vamos começar criando um novo produto.

Para criar um novo produto, vamos executar o seguinte `curl` :

```
curl --location --request POST
'http://localhost:8080/produto' \
--header 'Content-Type: application/json' \
--data-raw '{
  "id":0,
  "descricao": "teste1234",
  "preco": 1.23
}'
```

Após a execução, receberemos esta mensagem no terminal, indicando que o nosso produto foi criado com sucesso:

```
{"sucesso":true}
```

Criaremos mais um produto, apenas para melhorar os nossos próximos testes:

```
curl --location --request POST
'http://localhost:8080/produto' \
--header 'Content-Type: application/json' \
--data-raw '{
  "id":0,
  "descricao": "meuproduto23legal",
```

```
  "preco": 1.03
}'
```

Vamos testar nossa busca por produto? Utilizaremos a busca por descrição para simular uma busca por produtos. Vamos executar o seguinte `curl` :

```
curl --location --request GET
'http://localhost:8080/produto/descricao/23'
```

Na nossa chamada, utilizamos para a pesquisa o valor *23*, que é uma parte da descrição de ambos os produtos. O retorno, como esperado, é uma lista com os dois produtos:


```
{
    "produto": [
        {
            "descricao": "teste1234",
            "id": 1,
            "preco": 1.23
        },
        {
            "descricao": "meuproduto23legal",
            "id": 2,
            "preco": 1.03
        }
    ]
}
```


Vamos agora criar um cliente, que fará pedidos no nosso e-commerce. Para isso, executamos o seguinte `curl` :

```
curl --location --request POST
'http://localhost:8080/cliente' \
--header 'Content-Type: application/json' \
--data-raw '{
  "detalhe": {
    "id": 0,
    "nome": "Alexandre"
  },
  "enderecos": [
    {
      "id": 0,
      "rua":"rua teste 1",
      "numero":390,
      "cidade":"Sao Paulo",
      "estado":"SP",
      "cep":"12332456"
    },
    {
      "id": 0,
      "rua":"rua teste 2",
      "numero":158,
      "cidade":"Sao Paulo",
      "estado":"SP",
      "cep":"46756888"
    }
    ]

}
```

Após a execução, receberemos esta mensagem no terminal, indicando que o nosso cliente foi criado com sucesso:

```
{"sucesso":true}
```

Oops, o cliente cometeu um erro em seu cadastro! Um de seus endereços fica na *rua teste123*, não na *rua teste 2*. Para efetuar a correção, vamos chamar o seguinte `curl`, que efetua uma operação de `PATCH` na API:

```
curl --location --request PATCH
'http://localhost:8080/cliente' \
--header 'Content-Type: application/json' \
--data-raw '{
  "detalhe": {
    "id": 1,
    "nome": "Alexandre"
  },
  "enderecos": [
    {
      "id": 2,
      "rua":"rua teste 123",
      "numero":158,
      "cidade":"Sao Paulo",
      "estado":"SP",
      "cep":"46756888"
    }
    ]

}'
```

Após a execução, receberemos esta mensagem no terminal, indicando que o nosso cliente foi alterado com sucesso:

```
{"sucesso":true}
```

Agora, vamos fazer o nosso cliente efetuar uma compra. O seguinte comando efetua uma compra em nome do nosso cliente recém-criado:

```
curl --location --request POST
'http://localhost:8080/pedido' \
--header 'Content-Type: application/json' \
--data-raw '{
  "pedido": {
    "descricao": "pedido 1",
    "clienteId": 1
  },
  "produtos": [
    {
      "produtoId": 1,
      "quantidade": 3
    },
    {
      "produtoId": 2,
      "quantidade": 4
    }
    ]
}'
```

Após a execução, receberemos esta mensagem no terminal, indicando que o nosso pedido foi alterado com sucesso:

```
{"sucesso":true}
```

Para finalizar, vamos observar os dados nas tabelas, para confirmar que tudo foi persistido corretamente. Na tabela de produtos, temos os nossos produtos:

| | id<br>[PK] bigint | descricao<br>character varying | preco<br>double precision |
|---|---|---|---|
| 1 | 1 | teste1234 | 1.23 |
| 2 | 2 | meuproduto23legal | 1.03 |
| | | | |

Figura 6.2: Tabela de produtos.

Nas tabelas de clientes e endereços, temos o nosso cliente, com os endereços devidamente atualizados:

| | id<br>[PK] bigint | nome<br>character varying |
|---|---|---|
| 1 | 1 | Alexandre |
| | | |

Figura 6.3: Tabela de clientes.

| | id [PK] bigint | rua character varying | numero bigint | cidade character varying | estado character varying | cep character varying | cliente_id bigint |
|---|---|---|---|---|---|---|---|
| 1 | 1 | rua teste 1 | 390 | Sao Paulo | SP | 12332456 | 1 |
| 2 | 2 | rua teste 123 | 158 | Sao Paulo | SP | 46756888 | 1 |
| | | | | | | | |

Figura 6.4: Tabela de endereços.

Na tabela de pedidos, temos o cabeçalho do nosso pedido:

| | id [PK] bigint | descricao character varying | cliente_id bigint |
|---|---|---|---|
| 1 | 1 | pedido 1 | 1 |
| | | | |

Figura 6.5: Tabela de pedidos.

Com os seus respectivos itens:

| | pedido_id bigint | produto_id bigint | quantidade bigint |
|---|---|---|---|
| 1 | 1 | 1 | 3 |
| 2 | 1 | 2 | 4 |

Figura 6.6: Tabela de itens de pedidos.

Sucesso! Temos nossa API completa e funcional!

No capítulo 1, tivemos o famigerado problema desta API acessando a API de controle de estoque. Vamos implementar esse acesso usando um servidor de mocks para simular a outra API.

## 6.3 Acessando uma API externa

Incluiremos agora a consulta à API de estoque. Na criação de pedidos, vamos montar uma validação, na qual, para cada item do pedido, consultamos a sua disponibilidade. Se algum dos pedidos não estiver disponível, ou um erro de acesso à API ocorrer, não criamos o pedido e retornamos o HTTP Status 400 para o consumidor do endpoint.

Para simular a API de estoque, utilizaremos o `Mockoon`. Para instalá-lo, basta baixar o instalador na página https://mockoon.com/download/.

Vamos criar dois mocks, um para consultar o estoque do produto de id 1 e outro para o produto de id 2. Para o produto 1, retornaremos que ele está disponível, mas o produto 2 não. As imagens a seguir ilustram os mocks, com os JSONS das respostas que utilizaremos:

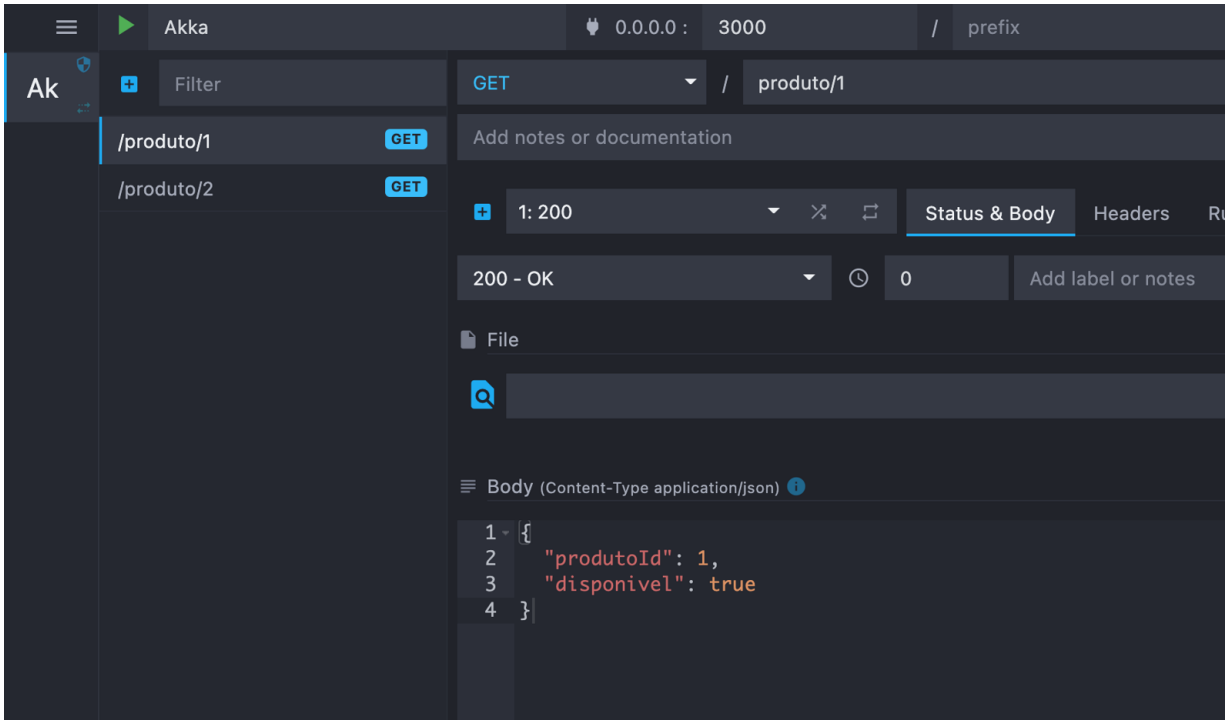



Figura 6.7: Mock de request do produto de id 1.

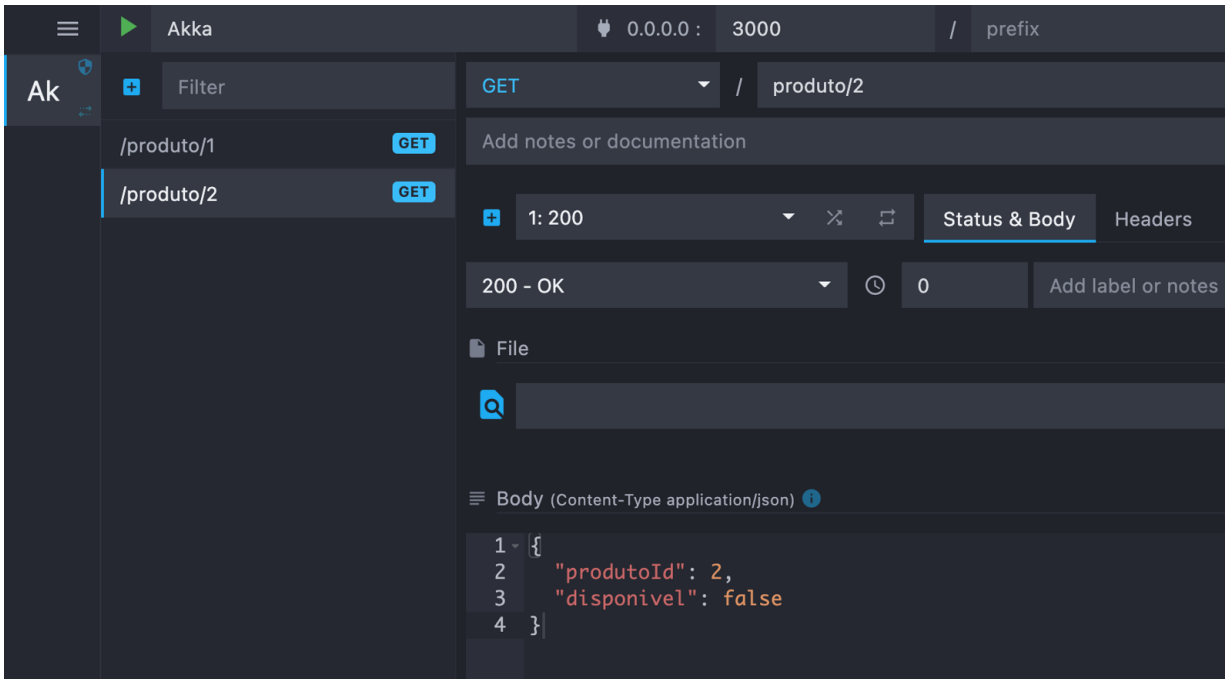



Figura 6.8: Mock de request do produto de id 2.

Para executar o servidor de mocks, basta clicar no botão verde no topo do programa.

Começaremos criando o ator de estoque. Esse ator utilizará a API de chamadas HTTP do Akka HTTP, onde assumimos o papel de cliente ao invés do papel de servidor, como fizemos no restante da API. Este é o código do nosso novo ator:

```
package com.casadocodigo.service

import akka.NotUsed
import akka.actor.typed.{ActorRef, Behavior,
SupervisorStrategy}
import akka.actor.typed.scaladsl.Behaviors
import akka.event.slf4j.Logger
```

```
import akka.http.scaladsl.Http
import com.casadocodigo.Boot.{executionContext, system}
import akka.http.scaladsl.model._
import akka.stream.scaladsl.{Sink, Source}
import akka.util.ByteString
import com.casadocodigo.route.SerializadorJSON
import spray.json.JsonParser

import scala.util.Success

object ServicoDeEstoque extends SerializadorJSON {

  val logger = Logger("ServicoDeEstoque")

  trait MensagemEstoque

  case class ConsultarEstoque(produtoId: Long, replyTo:
ActorRef[MensagemEstoque])

  case class RespostaConsultaEstoque(produtoId: Long,
disponivel: Boolean) extends MensagemEstoque

  case class RespostaConsultaEstoqueFalha() extends
MensagemEstoque

  def apply(): Behavior[ConsultarEstoque] =
Behaviors.supervise[ConsultarEstoque](behavior())
    .onFailure[Exception](SupervisorStrategy.restart)

  def behavior(): Behavior[ConsultarEstoque] =
Behaviors.receive {
    (_, mensagem) =>
      mensagem match {
        case ConsultarEstoque(produtoId, replyTo) =>
          Source.single((HttpRequest(uri =
s"http://localhost:3000/produto/${produtoId}"),
NotUsed))
```

```
            .via(Http().superPool[NotUsed]())
            .map {
              case (Success(res), _) =>

res.entity.dataBytes.runFold(ByteString(""))(_ ++
_).foreach { body =>
                  replyTo !
JsonParser(body.utf8String).convertTo[RespostaConsultaE
stoque]
                }
                res.discardEntityBytes()
              case _ => logger.error(f"erro ao chamar o
servico de estoque")
                replyTo !
RespostaConsultaEstoqueFalha()
            }
            .runWith(Sink.ignore)
      }
      Behaviors.same
  }

}
```

Precisamos deserializar a resposta da API do formato JSON para a classe Scala que representa essa resposta (a classe `RespostaConsultaEstoque`). Para fazer isso, voltamos a atualizar o nosso serializador incluindo a classe. Esta é a versão final do nosso serializador:

```
package com.casadocodigo.route

import
akka.http.scaladsl.marshallers.sprayjson.SprayJsonSuppo
rt
```

```
import com.casadocodigo.repository.{Cliente, Endereco,
Pedido, PedidoProduto, Produto}
import com.casadocodigo.route.Requisicoes.
{RequisicaoCliente, RequisicaoClienteDetalhe,
RequisicaoClienteEndereco, RequisicaoPedido,
RequisicaoPedidoDetalhe, RequisicaoPedidoItem}
import com.casadocodigo.route.Respostas.
{RespostaBuscaClienteSucesso,
RespostaBuscaPedidoSucesso,
RespostaBuscaProdutoSucesso, RespostaSucesso}
import
com.casadocodigo.service.ServicoDeEstoque.RespostaConsu
ltaEstoque
import spray.json.DefaultJsonProtocol

trait SerializadorJSON extends SprayJsonSupport with
DefaultJsonProtocol {

  implicit val respostaProdutoFormat =
jsonFormat3(Produto)
  implicit val respostaPedidoFormat =
jsonFormat3(Pedido)
  implicit val respostaClienteFormat =
jsonFormat2(Cliente)
  implicit val respostaEnderecoFormat =
jsonFormat7(Endereco)
  implicit val respostaPedidoProdutoFormat =
jsonFormat3(PedidoProduto)
  implicit val requisicaoPedidoDetalheFormat =
jsonFormat2(RequisicaoPedidoDetalhe)
  implicit val requisicaoPedidoItemFormat =
jsonFormat2(RequisicaoPedidoItem)
  implicit val requisicaoPedidoFormat =
jsonFormat2(RequisicaoPedido)
  implicit val requisicaoClienteDetalheFormat =
jsonFormat2(RequisicaoClienteDetalhe)
  implicit val requisicaoClienteEnderecoFormat =
```

```
jsonFormat6(RequisicaoClienteEndereco)
  implicit val requisicaoClienteFormat =
jsonFormat2(RequisicaoCliente)
  implicit val respostaSucessoFormat =
jsonFormat1(RespostaSucesso)
  implicit val respostaProdutoBuscarPorIdFormat =
jsonFormat1(RespostaBuscaProdutoSucesso)
  implicit val respostaPedidoBuscarPorIdFormat =
jsonFormat1(RespostaBuscaPedidoSucesso)
  implicit val respostaClienteBuscarPorIdFormat =
jsonFormat1(RespostaBuscaClienteSucesso)
  implicit val respostaConsultaDeEstoque =
jsonFormat2(RespostaConsultaEstoque)


}
```

Por fim, precisamos refatorar a nossa rota de criação de pedidos, construindo a validação de que falamos anteriormente. Esta é a nova versão do `criarPedido` de que necessitamos:

```
...
private def lift[T](futures: Seq[Future[T]]):
Seq[Future[Try[T]]] =
    futures.map(_.map {
      Success(_)
    }.recover { case t => Failure(t) })

  def waitAll[T](futures: Seq[Future[T]]):
Future[Seq[Try[T]]] =
    Future.sequence(lift(futures))

  def criarPedido(): Route = post {
    path("pedido") {
      entity(as[RequisicaoPedido]) { ped =>
```

```
        val hasOutOfStock =
Await.result(waitAll(ped.produtos.map(item =>
          atorDeEstoque.ask(ref =>
            ConsultarEstoque(item.produtoId, ref)
          )
        )), config.getInt("timeout") seconds).map(
          msg =>
            msg.getOrElse(RespostaConsultaEstoque(0,
disponivel = true))
        ).filter(msg =>
          msg match {
            case estoque: RespostaConsultaEstoque =>
              !estoque.disponivel
            case _ =>
              true
          }
        )

        if (hasOutOfStock.nonEmpty) {
          gerarRespostaDeErroDeEstoque()
        } else {
          gerarPedido(ped)
        }
      }
    }
  }

  private def gerarRespostaDeErroDeEstoque(): Route = {
    complete(BadRequest)
  }

  private def gerarPedido(ped: RequisicaoPedido): Route
= {
    val response:
Future[ServicoDePedidos.RespostaPedido] =
atorDePedidos.ask(ref =>
      MensagemCriarPedido(Pedido(0,
```

```
ped.pedido.descricao, ped.pedido.clienteId),
        ped.produtos.map({ item =>
          PedidoProduto(0, item.produtoId,
item.quantidade)
        }), ref))
    onComplete(response) {
      case Success(response) => response match {
        case RespostaGerenciamentoDePedido() =>
complete(RespostaSucesso(true))
        case _ => complete(BadRequest)
      }
      case Failure(e) => failWith(e)
    }
  }
...
```

Nenhuma outra alteração é necessária. Vamos reiniciar a nossa aplicação e testar a criação de um novo pedido através do seguinte comando `curl` :

```
curl --location --request POST
'http://localhost:8080/pedido' \
--header 'Content-Type: application/json' \
--data-raw '{
  "pedido": {
    "descricao": "pedido 2",
    "clienteId": 1
  },
  "produtos": [
    {
      "produtoId": 1,
      "quantidade": 3
    },
    {
      "produtoId": 2,
      "quantidade": 4
```

```
    }
    ]
}
```

Ao efetuar a chamada, recebemos como resposta o status HTTP 400 e a mensagem padrão do status no Akka HTTP:

```
The request contains bad syntax or cannot be fulfilled.
```

Vamos testar o que acontece se tentarmos criar o pedido apenas com produtos em estoque mudando o comando `curl` para o seguinte:

```
curl --location --request POST
'http://localhost:8080/pedido' \
--header 'Content-Type: application/json' \
--data-raw '{
  "pedido": {
    "descricao": "pedido 2",
    "clienteId": 1
  },
  "produtos": [
    {
      "produtoId": 1,
      "quantidade": 3
    }
    ]
}'
```

Dessa vez, recebemos uma resposta de sucesso, pois a nossa validação de estoque não apresentou nenhum problema:

```
{"sucesso":true}
```

E assim terminamos o desenvolvimento da API. Antes de avançarmos para outros tópicos, vamos construir um script de testes de carga da nossa aplicação. Esses testes serão utilizados para que possamos testar a resiliência da nossa aplicação e compreender por que a nossa API possui mais capacidade para resistir ao problema que ocorreu com a API de e-commerce do capítulo 1.

## 6.4 Medindo a robustez da nossa aplicação

Agora que temos a nossa API, vamos fazer alguns *testes de carga*. Testes de carga têm por objetivo validar a robustez da aplicação, avaliando quanto a nossa aplicação pode suportar antes de começar a rejeitar requisições, bem como identificar possíveis gargalos, como verificar o que acontece se uma dependência externa da aplicação está fora (como a API de estoque, por exemplo).

Para isso, vamos utilizar a biblioteca `Gatling`. Essa biblioteca, feita em Scala, nos permite criar *cenários* onde configuramos séries de chamadas HTTP, representando grupos de usuários invocando a aplicação. Podemos ter cenários de execução, como executar vários usuários por segundo durante

um período, ou uma quantidade de usuários crescente até um limite máximo.

Para configurar o `Gatling`, utilizaremos um `plugin` do sbt. Para isso, é necessário modificar o arquivo `plugins.sbt`:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager"
% "1.8.0")
addSbtPlugin("io.gatling" % "gatling-sbt" % "3.2.2")
```

E modificar o arquivo `build.sbt` configurando o `plugin` e adicionando as dependências do `Gatling`:

```
name := "akka-api"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"
val akkaHttpVersion = "10.2.6"
val slickVersion = "3.3.3"

dockerExposedPorts ++= Seq(8080)

enablePlugins(JavaAppPackaging)
enablePlugins(GatlingPlugin)

mainClass in Compile := Some("com.casadocodigo.Boot")

//Akka e outras dependencias
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
```

```
AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test,
  "com.typesafe.akka" %% "akka-http" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-spray-json" %
akkaHttpVersion
)

// dependencias de logging
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)

// dependencias de banco
libraryDependencies ++= Seq(
  "com.typesafe.slick" %% "slick" % slickVersion,
  "com.typesafe.slick" %% "slick-hikaricp" %
slickVersion,
  "org.postgresql" % "postgresql" % "42.2.23"
)

libraryDependencies ++= Seq(
  "io.gatling.highcharts" % "gatling-charts-highcharts"
% "3.6.1" % "test",
  "io.gatling" % "gatling-test-framework"    % "3.6.1"
% "test"
)
```

Antes de construir os nossos testes, executaremos um passo para facilitar a visualização das informações. Vamos criar um arquivo chamado `logback.xml` dentro da pasta `resources` com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <logger name="scala.slick" level="INFO" />
</configuration>
```

O objetivo dessa configuração é limitar os logs do Slick para o nível `INFO` a fim de permitir que enxerguemos os logs do Gatling mais facilmente, sem a quantidade enorme de logs que o Slick gera.

Escreveremos agora os testes do Gatling. Nossos testes serão criados dentro de traits, onde teremos diferentes cenários de execução. Vamos começar criando os cenários para os testes de produtos:

```
package com.casadocodigo.stress.simulations

import io.gatling.core.Predef._
import io.gatling.core.structure.ScenarioBuilder
import io.gatling.http.Predef._

trait SimulacaoDeProdutos extends UtilitarioDeNumeros {

  val cenarioProduto: ScenarioBuilder =
scenario("CenarioDeProdutos")
    .exec(http("request_post")
      .post("/produto").body(StringBody(
      s"""{ "id":0,
         |  "descricao": "teste$generateLong",
         |  "preco": 1.23 }""".stripMargin)).asJson)
    .exec(http("request_patch")
      .patch("/produto").body(StringBody(
      s"""{ "id":1,
         |  "descricao": "teste$generateLong",
         |  "preco": 1.23 }""".stripMargin)).asJson)
```

```
    .exec(http("request_get")
      .get("/produto/descricao/teste"))


}
```

Nesse teste, configuramos a criação de um produto, sua alteração e uma consulta por descrição na sequência.

Em seguida, vamos criar os cenários para os testes de clientes:

```
package com.casadocodigo.stress.simulations

import io.gatling.core.Predef._
import io.gatling.core.structure.ScenarioBuilder
import io.gatling.http.Predef._

trait SimulacaoDeClientes extends UtilitarioDeNumeros {

  val cenarioPrimeiroCliente: ScenarioBuilder =
scenario("CenarioDePrimeiroClientes")
    .exec(http("request_post")
      .post("/cliente").body(StringBody(
      s"""{
         |  "detalhe": {
         |    "id": 0,
         |    "nome": "Alexandre"
         |  },
         |  "enderecos": [
         |    {
         |      "id": 0,
         |      "rua":"rua teste 1",
         |      "numero":390,
         |      "cidade":"Sao Paulo",
```

```
            |      "estado":"SP",
            |      "cep":"12332456"
            |    },
            |    {
            |      "id": 0,
            |      "rua":"rua teste 2",
            |      "numero":158,
            |      "cidade":"Sao Paulo",
            |      "estado":"SP",
            |      "cep":"46756888"
            |    }
            |    ]
            |
            |}""".stripMargin)).asJson)

  val cenarioCliente: ScenarioBuilder =
scenario("CenarioDeClientes")
    .exec(http("request_post")
      .post("/cliente").body(StringBody(
      s"""{
          |  "detalhe": {
          |    "id": 0,
          |    "nome": "Alexandre"
          |  },
          |  "enderecos": [
          |    {
          |      "id": 0,
          |      "rua":"rua teste 1",
          |      "numero":390,
          |      "cidade":"Sao Paulo",
          |      "estado":"SP",
          |      "cep":"12332456"
          |    },
          |    {
          |      "id": 0,
          |      "rua":"rua teste 2",
          |      "numero":158,
```

```
            |      "cidade":"Sao Paulo",
            |      "estado":"SP",
            |      "cep":"46756888"
            |    }
            |    ]
            |
            |}""".stripMargin)).asJson)
      .exec(http("request_patch")
        .patch("/cliente").body(StringBody(
        s"""{
            |  "detalhe": {
            |    "id": 1,
            |    "nome": "Alexandre"
            |  },
            |  "enderecos": [
            |    {
            |      "id": 2,
            |      "rua":"rua teste 123",
            |      "numero":158,
            |      "cidade":"Sao Paulo",
            |      "estado":"SP",
            |      "cep":"46756888"
            |    }
            |    ]
            |
            |}""".stripMargin)).asJson)

}
```

Nessa trait, é possível observar que temos dois cenários configurados, `cenarioPrimeiroCliente` e `cenarioCliente` . A razão para isso é que, nos nossos testes, primeiro executaremos todos os testes para apenas um usuário. Isso porque, caso as tabelas não existam, nossa aplicação as criará na

primeira execução. Esse é o nosso caso (mais à frente, criaremos um script de execução dos testes, onde sempre criamos um novo banco para fazer os testes), então realizaremos esse passo de modo que os testes de carga sejam feitos com a base já devidamente criada, que é o cenário normal em que a aplicação estará operando.

Vamos agora criar a trait com os cenários de testes para pedidos:

```
package com.casadocodigo.stress.simulations

import io.gatling.core.Predef._
import io.gatling.core.structure.ScenarioBuilder
import io.gatling.http.Predef._

import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

trait SimulacaoDePedidos extends UtilitarioDeNumeros {

  val cenarioPrimeiroPedido: ScenarioBuilder =
scenario("CenarioDocenarioPrimeiroPedido")
    .exec(http("request_post")
      .post("/pedido").body(StringBody(
      s"""{
         |  "pedido": {
         |    "descricao": "pedido 1",
         |    "clienteId": 1
         |  },
         |  "produtos": [
         |    {
         |      "produtoId": 1,
         |      "quantidade": 3
```

```
          |    }
          |    ]
          |}""".stripMargin)).asJson)
    .pause(2 second)

  val cenarioPedido: ScenarioBuilder =
scenario("CenarioDePedidos")
    .exec(http("request_post")
      .post("/pedido").body(StringBody(
      s"""{
          |  "pedido": {
          |    "descricao": "pedido 1",
          |    "clienteId": 1
          |  },
          |  "produtos": [
          |    {
          |      "produtoId": 1,
          |      "quantidade": 3
          |    }
          |    ]
          |}""".stripMargin)).asJson)

}
```

Nessa trait, ao final do cenário de execução inicial, inserimos uma pequena pausa de dois segundos, através do comando `.pause(2 second)`. Isso é feito apenas como uma dupla garantia de que só vamos iniciar os testes de carga com a API devidamente inicializada e estabilizada.

Assim como no caso dos cenários de clientes, aqui temos também um cenário de execução inicial, para a criação das tabelas de pedidos.

Em todas as traits, realizamos a utilização de outra trait, chamada `UtilitarioDeNumeros`. Essa trait consiste em um método utilitário, com o qual geramos números randômicos. Isso é utilizado para introduzir um pouco de variação nos nossos dados de testes, a fim de não inserirmos sempre as mesmas coisas. Este é o código dessa trait utilitária:

```
package com.casadocodigo.stress.simulations

trait UtilitarioDeNumeros {

  val leftLimit = 1L
  val rightLimit = 100L

  def generateLong: Long = leftLimit + (Math.random *
(rightLimit - leftLimit)).toLong

}
```

Todas essas chamadas HTTP possuem configurações em comum, como o host e a porta a serem chamados, bem como o `encoding` das requisições (JSON, neste caso). Para isso, criamos uma trait chamada `Protocolo` com o seguinte código:

```
package com.casadocodigo.stress.simulations

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.http.protocol.HttpProtocolBuilder

trait Protocolo {

  val protocolo: HttpProtocolBuilder = http
    .baseUrl("http://localhost:8080")
    .acceptHeader("application/json")

}
```

Agora, criaremos o nosso executor. Nosso executor será implementado pela classe `ExecutorDeSimulacoes`. Essa classe estende a classe `Simulation` do `Gatling`, além de todas as traits criadas anteriormente. Nele, vamos realizar o `setup` da nossa execução, que consiste da seguinte sequência de passos:

- Executa o cenário de produtos 1 vez.
- Executa o cenário de clientes (inicial) 1 vez.
- Executa o cenário de pedidos 1 (inicial) vez.
- Executa o cenário de clientes por um período de 2 minutos, distribuindo usuários até atingir um pico de 300 usuários.
- Executa o cenário de pedidos por um período de 4 minutos, com uma quantidade constante de 50 novos usuários por segundo.

O código a seguir cria o nosso executor, com a sequência de execuções supracitada:

```
package com.casadocodigo.stress.simulations

import io.gatling.core.Predef._
import scala.concurrent.duration._
import scala.language.postfixOps

class ExecutorDeSimulacoes extends Simulation with
SimulacaoDeProdutos with SimulacaoDePedidos with
SimulacaoDeClientes with Protocolo {

  setUp(
    cenarioProduto.inject(atOnceUsers(1)),
    cenarioPrimeiroCliente.inject(atOnceUsers(1)),
    cenarioPrimeiroPedido.inject(atOnceUsers(1)),
    cenarioCliente.inject(rampUsers(300) during (2
minutes)),
    cenarioPedido.inject(constantUsersPerSec(50) during
(4 minutes))
  ).protocols(protocolo)

}
```

Continuando, faremos uma pequena modificação no código da nossa aplicação. No nosso ator de estoque, deixamos o `host` da API referenciado como `localhost`. Essa referência não funcionará quando estivermos executando a API em modo dockerizado, pois, nesse cenário, `localhost` se refere a tentar chamar algo que está dentro do próprio contêiner.

Para resolver isso, utilizaremos o endereço reservado do docker-compose chamado `host.docker.internal`. Vamos modificar nosso ator `ServicoDeEstoque` para o seguinte código:

```
package com.casadocodigo.service

import akka.NotUsed
import akka.actor.typed.{ActorRef, Behavior,
SupervisorStrategy}
import akka.actor.typed.scaladsl.Behaviors
import akka.event.slf4j.Logger
import akka.http.scaladsl.Http
import com.casadocodigo.Boot.{config, executionContext,
system}
import akka.http.scaladsl.model._
import akka.stream.scaladsl.{Sink, Source}
import akka.util.ByteString
import com.casadocodigo.route.SerializadorJSON
import spray.json.JsonParser

import scala.util.Success

object ServicoDeEstoque extends SerializadorJSON {

  val logger = Logger("ServicoDeEstoque")

  trait MensagemEstoque

  case class ConsultarEstoque(produtoId: Long, replyTo:
ActorRef[MensagemEstoque])

  case class RespostaConsultaEstoque(produtoId: Long,
disponivel: Boolean) extends MensagemEstoque

  case class RespostaConsultaEstoqueFalha() extends
```

```
MensagemEstoque

  def apply(): Behavior[ConsultarEstoque] =
Behaviors.supervise[ConsultarEstoque](behavior())
    .onFailure[Exception](SupervisorStrategy.restart)

  def behavior(): Behavior[ConsultarEstoque] =
Behaviors.receive {
    (_, mensagem) =>
      mensagem match {
        case ConsultarEstoque(produtoId, replyTo) =>
          Source.single((HttpRequest(uri =
s"http://${config.getString("url")}:3000/produto/$produ
toId"), NotUsed))
            .via(Http().superPool[NotUsed]())
            .map {
              case (Success(res), _) =>

res.entity.dataBytes.runFold(ByteString(""))(_ ++
_).foreach { body =>
                  replyTo !
JsonParser(body.utf8String).convertTo[RespostaConsultaE
stoque]
                }
                res.discardEntityBytes()
              case _ => logger.error(f"erro ao chamar o
servico de estoque")
                replyTo !
RespostaConsultaEstoqueFalha()
            }
            .runWith(Sink.ignore)
      }
      Behaviors.same
  }

}
```

A seguir, modificaremos o arquivo `application.base.conf` adicionando o valor default da propriedade `url` :

```
timeout = 5
url="localhost"

# Datasources
database.postgres {
  connectionPool = "HikariCP"
  dataSourceClass =
"org.postgresql.ds.PGSimpleDataSource"
  properties = {
    serverName = "localhost"
    serverName = ${?DB_HOST}
    portNumber = "5432"
    databaseName = "ecommerce"
    user = "postgres"
    password = "teste"
    password = ${?DB_PASS}
  }
  numThreads = 10
}
```

E vamos modificar o arquivo `prod.conf` a fim de configurar a propriedade para execução em modo dockerizado:

```
include "application.base.conf"

url="host.docker.internal"
akka.loglevel = "INFO"
```

Finalizando a nossa preparação para os testes, vamos criar um *shell script* chamado `run_load_tests.sh`. Neste script, publicamos uma imagem Docker da nossa API, recriamos nosso banco e API com a nova versão e executamos os testes:

```
#!/usr/bin/env bash

sbt docker:publishLocal

docker-compose down
docker-compose up -d

sleep 10

sbt gatling:test
```

Ufa! Finalmente estamos prontos para começar! Vamos executar nossos testes de carga utilizando o *shell script*:

```
./run_load_tests.sh
```

Durante a execução, podemos observar um "quadro" no terminal, mostrando a evolução da execução. As quantidades de requisições são divididas entre OK (sucesso) e KO (falha):

```
================================================================================
2021-08-23 21:50:10
```

```
230s elapsed
---- Requests ------------------------------------
-------------------------
> Global
(OK=12105  KO=0     )
> request_post
(OK=11803  KO=0     )
> request_patch
(OK=301    KO=0     )
> request_get
(OK=1      KO=0     )

---- CenarioDePrimeiroClientes -----------------------
-------------------------
[##########################################################
####################]100%
          waiting: 0      / active: 0      / done: 1
---- CenarioDeProdutos --------------------------------
-------------------------
[##########################################################
####################]100%
          waiting: 0      / active: 0      / done: 1
---- CenarioDePedidos ---------------------------------
-------------------------
[##########################################################
################-   ] 95%
          waiting: 499    / active: 1      / done:
11500
---- CenarioDocenarioPrimeiroPedido -------------------
-------------------------
[##########################################################
####################]100%
          waiting: 0      / active: 0      / done: 1
---- CenarioDeClientes --------------------------------
-------------------------
[##########################################################
####################]100%
```

```
          waiting: 0      / active: 0      / done: 300
================================================================
=========================
```

No final da execução, temos os valores finais da execução:

```
================================================================
=========================
---- Global Information --------------------------------
------------------------
> request count
12605 (OK=12605  KO=0     )
> min response time
7 (OK=7      KO=-     )
> max response time
2253 (OK=2253   KO=-     )
> mean response time
48 (OK=48     KO=-     )
> std deviation
219 (OK=219    KO=-     )
> response time 50th percentile
13 (OK=13     KO=-     )
> response time 75th percentile
15 (OK=15     KO=-     )
> response time 95th percentile
41 (OK=41     KO=-     )
> response time 99th percentile
1616 (OK=1616   KO=-     )
> mean requests/sec
52.521 (OK=52.521 KO=-     )
---- Response Time Distribution ------------------------
------------------------
> t < 800 ms
12388 ( 98%)
> 800 ms < t < 1200 ms
```

```
40 (  0%)
> t > 1200 ms
177 (  1%)
> failed
0 (  0%)
================================================================
========================
```

Como podemos ver, a aplicação recebeu 12.605 requisições, sendo que 98% delas levaram menos de 800 milissegundos para executar. Também é possível notar que, em média, as requisições levaram 48 milissegundos para executar.

O Gatling também possui uma bela interface gráfica, onde podemos ver os relatórios no formato HTML. Para visualizá-los, podemos navegar dentro da estrutura do projeto para o conjunto de pastas `/target/gatling` , onde podemos ver diferentes pastas que representam, cada uma, uma execução:

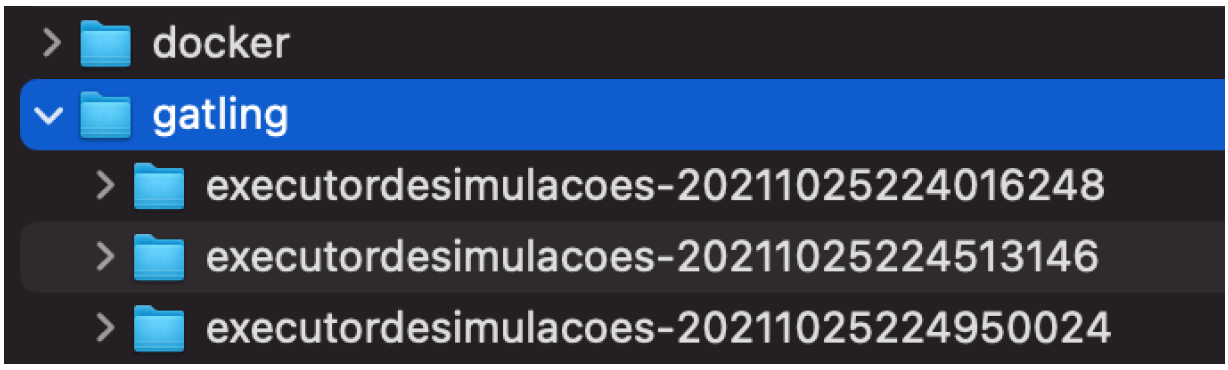



Figura 6.9: Pastas com relatórios do Gatling.

E abrindo os arquivos `index.html` dentro das pastas, temos acesso aos relatórios:

Figura 6.10: Relatório HTML do Gatling.

Faremos agora o seguinte teste: parar o servidor do `Mockoon`.

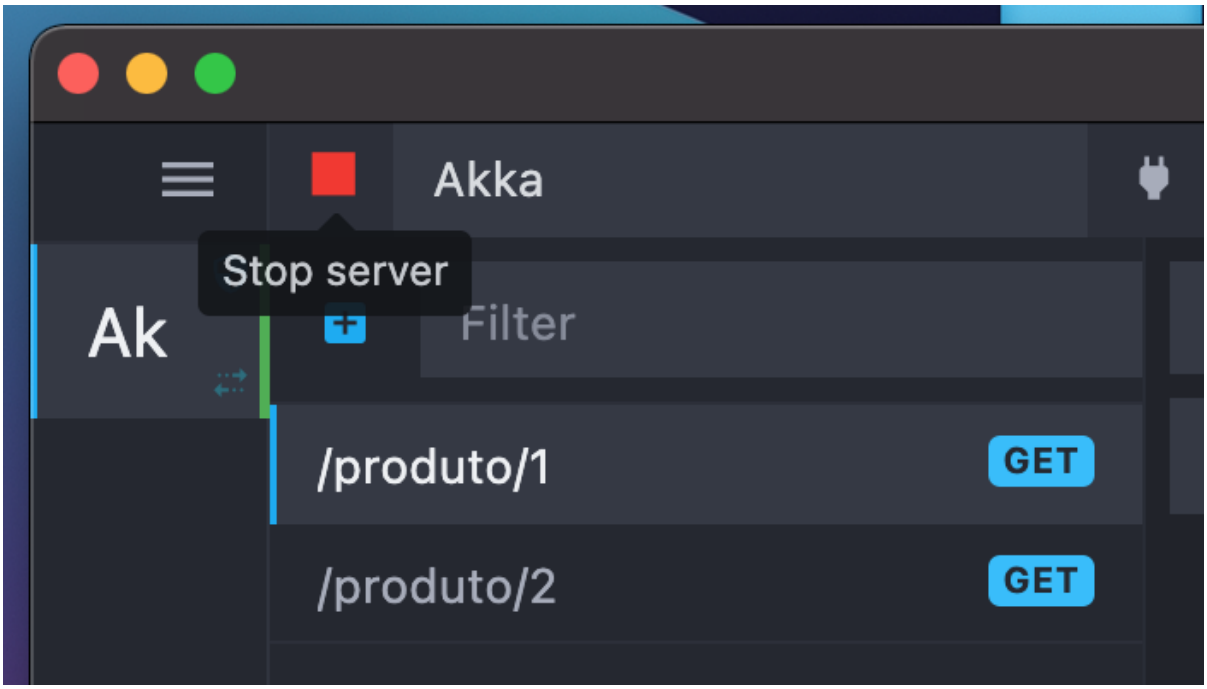



Figura 6.11: Parando o Mockoon.

E vamos reexecutar os testes. Isso simulará exatamente o cenário que causou a derrubada da API de e-commerce, no capítulo 1. Vamos ver o que ocorre, executando o *shell script*:

```
./run_load_tests.sh
```

Como é esperado, podemos ver que cerca de 95% das requisições falharam. Isso é esperado, dado que a maioria das requisições na execução são compostas por criações de pedidos, que falharam devido ao serviço de estoque estar fora do ar:

```
================================================================
========================
```

```
---- Global Information --------------------------------
-------------------------
> request count
12605 (OK=604    KO=12001 )
> min response time
2 (OK=11     KO=2     )
> max response time
6172 (OK=3003   KO=6172  )
> mean response time
50 (OK=67     KO=49    )
> std deviation
373 (OK=272    KO=378   )
> response time 50th percentile
5 (OK=18     KO=5     )
> response time 75th percentile
6 (OK=24     KO=6     )
> response time 95th percentile
19 (OK=180    KO=10    )
> response time 99th percentile
1584 (OK=1489   KO=1589  )
> mean requests/sec
52.303 (OK=2.506  KO=49.797)
---- Response Time Distribution ------------------------
-------------------------
> t < 800 ms
590 (  5%)
> 800 ms < t < 1200 ms
6 (  0%)
> t > 1200 ms
8 (  0%)
> failed
12001 ( 95%)
---- Errors --------------------------------------------
-------------------------
>
status.find.in(200,201,202,203,204,205,206,207,208,209,
304), f  11968 (99.73%)
```

```
ound 400
>
status.find.in(200,201,202,203,204,205,206,207,208,209,
304), f     33 ( 0.27%)
ound 500
================================================================
========================
```

Agora, o mais interessante é quando observamos a média do tempo das execuções: em média, as requisições levaram 50 milissegundos para executar, contra os 48 milissegundos no cenário em que o `Mockoon` estava ativo. Isso demonstra que a indisponibilidade do serviço de estoque pouco afetou a nossa API em termos de disponibilidade. Se tivéssemos uma API como esta no cenário do capítulo 1, nossa história teria sido diferente e um pouco mais feliz.

Com isso, concluímos nosso *tour* pelo Akka HTTP. Agora, vamos aprender sobre o Akka Streams e de que modo a sua abstração nos permite construir integrações entre sistemas utilizando conceitos de *streaming programming*.

CAPÍTULO 7

# Introdução ao Akka Streams

Com os nossos conhecimentos sobre o Akka solidificados, neste capítulo, vamos começar a aprender como desenvolver integrações entre sistemas utilizando Akka Streams. Integrações fazem parte do dia a dia de qualquer desenvolvedor, sempre precisamos desenvolver soluções que nos permitem transferir informações de um ponto A para um ponto B.

Conforme já dito no decorrer do livro, o Akka Streams foi construído em cima do Akka. Isso significa que, internamente, continuamos trabalhando com atores e que temos apenas uma nova camada na qual podemos construir aplicações seguindo o modelo de programação orientada a *streams* sem a necessidade de utilizarmos os atores diretamente.

## 7.1 Conceitos do Akka Streams

No modelo de programação orientada a streams - não utilizamos a tradução da palavra *streams* neste contexto, que seria algo como "correnteza", "riacho"

-, nossas aplicações são definidas como fluxos de dados que passam por diversas transformações e operações e seguem até um destino final.

Comumente é feita uma analogia de streams com rios, como a própria tradução da palavra sugere. Seguindo essa analogia, podemos dividir uma stream em:

- Nascente: é a origem dos dados, de onde obteremos esses dados para processar. Diferentes tipos de tecnologias podem ser utilizados como nascente, como tecnologias de mensageria, como o `Apache Kafka` ou `AWS SQS` , sistemas de arquivos, tabelas de banco de dados etc. Dentro do Akka Streams, essa etapa é chamada de `Source` .
- Correnteza/Curso: são as operações que compõem a stream, como transformações, filtragens, agregações etc. Veremos exemplos dessas operações no decorrer do capítulo. Na terminologia do Akka Streams, chamamos essas operações de `Flow` .
- Foz: a operação final da stream, o destino final dos dados após todas as operações serem concluídas. Assim como na nascente, diferentes tecnologias podem ser utilizadas como foz, como as já citadas tecnologias de mensageria,

arquivos etc. Na terminologia do Akka Streams, essa etapa é chamada de `Sink`.

O diagrama a seguir ilustra um exemplo real de uma stream com as etapas devidamente separadas:

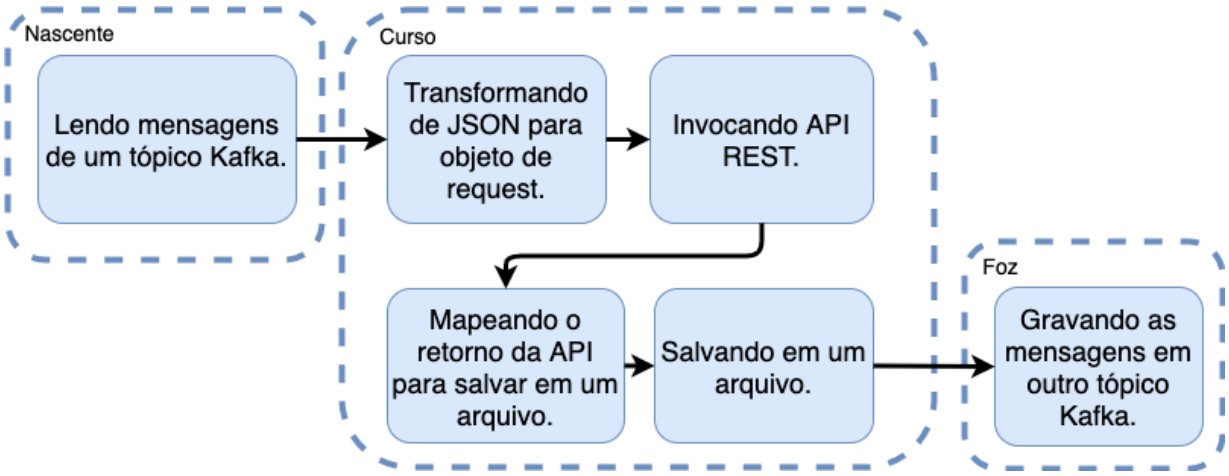

Nascente
Lendo mensagens de um tópico Kafka.
Curso
Transformando de JSON para objeto de request.
Invocando API REST.
Mapeando o retorno da API para salvar em um arquivo.
Salvando em um arquivo.
Foz
Gravando as mensagens em outro tópico Kafka.

Figura 7.1: Estrutura de uma stream.

No próximo capítulo, veremos um exemplo com os passos mostrados na imagem anterior executando em paralelo em vez de um depois do outro em sequência. Devido a essa estrutura de nós (passos) com ligações a partir de um nó raiz (neste caso, o nó raiz sendo a leitura de mensagens do tópico Kafka), dentro do Akka Streams, o conjunto de nós que compõe uma stream são definidos como uma estrutura de dados do tipo *grafo*. Para saber mais sobre grafos, sugiro a leitura do seguinte artigo na *Wikipédia*: https://pt.wikipedia.org/wiki/Teoria_dos_grafos

Todas essas alegorias e conceitos que vemos no Akka Streams nos permitem compreender por que esse modelo é ideal para o desenvolvimento de integrações: integrações, em linhas gerais, nada mais são do que aplicações que transportam dados de um sistema A para um sistema B, exatamente o que fazemos quando construímos uma stream.

**Back-pressure do Akka Streams**

Um conceito importante do Akka Streams é conhecido como *back-pressure*. Back-pressure - traduzido como "pressão contrária" - significa que, nas streams, o processamento ocorre de acordo

com a frequência de processamento das mensagens pelo consumidor da informação e não o contrário.

Por exemplo, na stream do diagrama descrito anteriormente, as chamadas para a API REST são feitas internamente por um ator, que vai consumindo mensagens oriundas do leitor de mensagens do tópico Kafka, executado por outro ator. Conforme já aprendemos, no Akka, toda comunicação entre os atores é feita de modo assíncrono, com mensagens sendo buferizadas entre os atores, que consomem suas mensagens conforme as suas capacidades.

Assim, se a API sendo chamada começar a processar as requisições mais lentamente, as mensagens serão buferizadas em vez de o ator ser "esmagado" pelo outro ator, que freneticamente está publicando as mensagens recebidas no tópico Kafka. O diagrama a seguir ilustra esses conceitos:

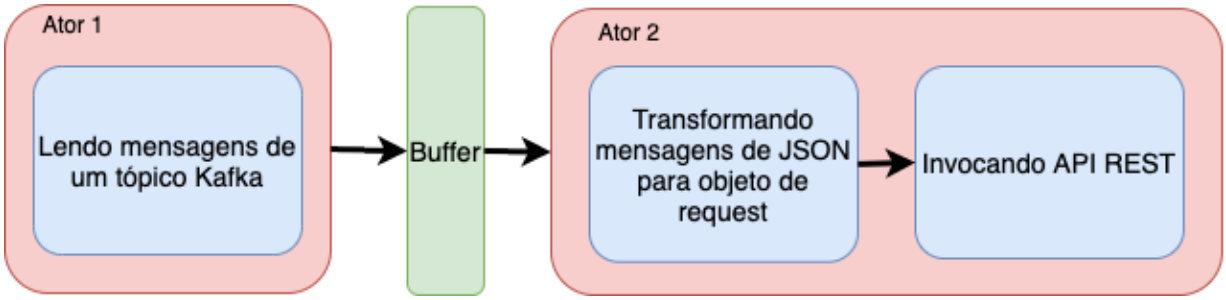



Figura 7.2: Back-pressure na prática.

Agora que temos os conceitos do Akka Streams bem definidos, vamos aprender a construir os nossos

sistemas utilizando Akka Streams.

## 7.2 Nossa primeira stream

Vamos começar a nossa construção criando uma simples stream que vai imprimir no console uma sequência de números, apenas para termos um primeiro vislumbre dos conceitos que vimos anteriormente. Para começar, vamos criar um novo projeto e configurar as nossas dependências criando o arquivo `build.sbt` da seguinte forma:

```
name := "akka-streams"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"

mainClass in Compile := Some("com.casadocodigo.Boot")

//Akka e outras dependencias
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test,
)

// dependencias de logging
libraryDependencies ++= Seq(
```

```
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)
```

Dependências definidas, vamos criar nossa primeira stream. É uma stream bastante simples, apenas uma sequência de 100 números gerada em memória, sendo impressa no terminal. Nossa stream vai executar dentro do objeto `Boot`, que inicializa a nossa aplicação. Para isso, vamos criar o objeto com o seguinte conteúdo:

```
package com.casadocodigo


import akka.{Done, NotUsed}
import akka.actor.ActorSystem
import akka.stream.scaladsl.Source

import scala.concurrent.{ExecutionContextExecutor,
Future}


object Boot extends App {

  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")
  implicit val ec: ExecutionContextExecutor =
system.dispatcher

  val source: Source[Int, NotUsed] = Source(1 to 100)
  val done: Future[Done] = source.runForeach(i =>
println(i))

  done.onComplete(_ => system.terminate())
```

```
}
```

Nosso código é bastante simples, basta criar um sistema de atores e utilizar a stream. Criamos a nascente da stream através do objeto `Source`, passando uma lista como parâmetro - o operador `to` é um gerador do Scala que nos permite criar listas a partir de parâmetros de início e fim da lista.

Em seguida, definimos o método `runForeach`. Esse método compreende tanto uma operação de curso da stream quanto de foz, devolvendo um `Sink`. O `Sink` é encapsulado em uma `Future`, que pode ser observada para aguardarmos os resultados da stream. Utilizando o método `onComplete`, encerramos a aplicação após o término da execução da stream.

Executando a aplicação, podemos ver que os números são impressos com sucesso, como podemos ver no fragmento a seguir:

```
...
[info] loading project definition from
/Users/alexandrelourenco/livro-akka-
streams/capitulo_7_8/project
[info] loading settings for project capitulo_7_8 from
build.sbt ...
[info] set current project to akka-streams (in build
file:/Users/alexandrelourenco/livro-akka-
```

```
streams/capitulo_7_8/)
[info] compiling 1 Scala source to
/Users/alexandrelourenco/livro-akka-
streams/capitulo_7_8/target/scala-2.13/classes ...
[info] running com.casadocodigo.Boot
1
2
3
4
5
6
7
8
9
10
...
```

## Fazendo algumas transformações com a stream

Vamos testar a realização de algumas operações em cima da stream. Vamos filtrar o processamento para utilizar apenas números pares e, em vez de imprimir apenas os números, vamos mapear os números para um texto e imprimir o texto por fim:

```
package com.casadocodigo


import akka.{Done, NotUsed}
import akka.actor.ActorSystem
import akka.stream.scaladsl.Source
import com.typesafe.config.{Config, ConfigFactory}

import scala.concurrent.Future
```

```
object Boot extends App {

  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")
  implicit val ec = system.dispatcher

  val source: Source[Int, NotUsed] = Source(1 to 100)
  val done: Future[Done] = source
    .filter(i => i % 2 == 0)
    .map(i => f"sou o numero $i")
    .runForeach(i => println(i))

  done.onComplete(_ => system.terminate())

}
```

Após a execução, observaremos uma sequência de frases `eu sou o numero X` apenas com números pares, mostrando que nossas transformações executaram com sucesso:

```
...
[info] loading project definition from
/Users/alexandrelourenco/livro-akka-
streams/capitulo_7_8/project
[info] loading settings for project capitulo_7_8 from
build.sbt ...
[info] set current project to akka-streams (in build
file:/Users/alexandrelourenco/livro-akka-
streams/capitulo_7_8/)
[info] compiling 1 Scala source to
/Users/alexandrelourenco/livro-akka-
streams/capitulo_7_8/target/scala-2.13/classes ...
[info] running com.casadocodigo.Boot
sou o numero 2
```

```
sou o numero 4
sou o numero 6
sou o numero 8
sou o numero 10
sou o numero 12
...
```

Assim concluímos a nossa introdução aos conceitos do Akka Streams. Agora que vimos o funcionamento básico dele, vamos avançar para o próximo capítulo, no qual veremos exemplos de streams utilizando tecnologias do mundo real.

## Capítulo 8

# Integrações do mundo real com Akka Streams

Neste capítulo, vamos estudar uma série de streams utilizando tecnologias do mundo real, como sistemas de arquivos, Apache Kafka, APIs REST, serviços da AWS, como S3, SQS etc. Esses exemplos nos permitirão ver como aplicar o Akka Streams em integrações reais do nosso dia a dia.

Para essas streams, reaproveitaremos o projeto que criamos no capítulo anterior, a nossa primeira stream, apenas adicionando as novas streams dentro do projeto. Vamos começar com uma integração que escuta arquivos `csv` em uma pasta, lendo as informações dos arquivos conforme são salvos em uma pasta e os enviando para um tópico Kafka.

Para facilitar o nosso desenvolvimento, utilizaremos conectores do projeto **Alpakka**. O projeto Alpakka consiste em um conjunto de vários conectores que podem ser utilizados como nascentes, cursos e fozes de streams, fornecendo meios para construir aplicações que utilizam as mais variadas tecnologias, como ElasticSearch, MongoDB, Sistemas de arquivos etc. Também possui suporte

pronto para utilizar os serviços dos principais provedores de computação em nuvem do mercado, como Google Cloud, AWS e Azure. A documentação do projeto pode ser encontrada em:

https://doc.akka.io/docs/alpakka/current/

## 8.1 Integrando um sistema de arquivos com o Apache Kafka

Vamos desenvolver nossa primeira stream que compõe um cenário do mundo real. Antes de começarmos, vamos apenas refatorar nossa primeira stream para um objeto a fim de facilitar a inserção de novas streams no nosso projeto.

Vamos começar criando um objeto chamado `PrimeiraStream` com o seguinte código:

```
package com.casadocodigo.streams

import akka.{Done, NotUsed}
import akka.stream.scaladsl.Source
import com.casadocodigo.Boot.{ec, system}

import scala.concurrent.Future

object PrimeiraStream {

  def primeiraStream(): Unit = {
    val source: Source[Int, NotUsed] = Source(1 to 100)
```

```
    val done: Future[Done] = source
      .filter(i => i % 2 == 0)
      .map(i => f"sou o numero $i")
      .runForeach(i => println(i))

    done.onComplete(_ => println("terminando a
execução!"))
  }

}
```

E modificar o objeto `Boot` da seguinte forma:

```
package com.casadocodigo


import akka.actor.ActorSystem
import com.casadocodigo.streams.PrimeiraStream

import scala.concurrent.ExecutionContextExecutor
import scala.language.postfixOps


object Boot extends App {

  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")
  implicit val ec: ExecutionContextExecutor =
system.dispatcher

  PrimeiraStream.primeiraStream()

}
```

Refatoração feita, vamos criar nossa stream de arquivos `csv` para tópicos Kafka.

## Criando o ambiente para a integração

Para começar, vamos modificar novamente o arquivo `build.sbt` adicionando as dependências do Alpakka. Os conectores do Alpakka possuem cada um sua própria dependência, permitindo incluir e excluir componentes com facilidade em uma arquitetura estilo *plug and play*:

```
name := "akka-streams"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"

mainClass in Compile := Some("com.casadocodigo.Boot")

//Akka e outras dependencias
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test
)

// dependencias de logging
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)

// dependencias do Alpakka
libraryDependencies ++= Seq(
```

```
  "com.lightbend.akka" %% "akka-stream-alpakka-file" %
"3.0.3",
  "com.typesafe.akka" %% "akka-stream-kafka" % "2.1.1"
)
```

A seguir, vamos criar o arquivo `docker-compose.yml` . Na nossa primeira stream, integraremos arquivos com o *Apache Kafka*, por isso criaremos um cluster Kafka para o exemplo. O código a seguir cria essa estrutura já automaticamente criando um tópico chamado `contas` na sua inicialização, que vamos utilizar na stream:

```
version: '3.9'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
    ports:
      - 2181:2181
  kafka:
    image: wurstmeister/kafka:2.12-2.5.0
    ports:
      - 9092:9092
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_CREATE_TOPICS: "contas:1:1"
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

O leitor pode se perguntar por que estamos criando também um *Zookeeper*. A arquitetura do Kafka se utiliza de um Zookeeper para algumas de suas operações, por isso precisamos criar esse contêiner também.

## Criando nossas streams

Agora, vamos criar as streams propriamente ditas. Criaremos várias streams neste exemplo a fim de tratar os diferentes cenários de processamento de arquivos, como inclusão e modificações de arquivos etc. Para isso, vamos criar um arquivo chamado `ArquivoParaKafka` e incluir o seguinte código:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.kafka.ProducerSettings
import akka.kafka.scaladsl.Producer
import akka.stream.alpakka.file.DirectoryChange
import akka.stream.alpakka.file.scaladsl.{Directory,
DirectoryChangesSource, FileTailSource}
import akka.stream.scaladsl.Source
import com.casadocodigo.Boot.system
import org.apache.kafka.clients.producer.ProducerRecord
import
org.apache.kafka.common.serialization.StringSerializer

import java.io.FileNotFoundException
import java.nio.file.{FileSystems, Path}
import scala.concurrent.duration.DurationInt
```

```
import scala.language.postfixOps

object ArquivoParaKafka {

  private val kafkaProducerSettings =
    ProducerSettings.create(system, new
StringSerializer(), new StringSerializer())
      .withBootstrapServers("localhost:9092")

  private val sistemaDeArquivos =
FileSystems.getDefault
  private val diretorio = "./input_dir"
  private val mudancasNoDiretorio =
DirectoryChangesSource(sistemaDeArquivos.getPath(direto
rio), pollInterval = 1 second, maxBufferSize = 1000)
  private val diretorioInicial: Source[Path, NotUsed] =
Directory.ls(sistemaDeArquivos.getPath(diretorio))

  def iniciarStreams(): Unit = {
    diretorioInicial.runForeach {
      path =>
        obterVerificadorDeArquivoDeletado(path)

obterLeitorDeArquivo(path).merge(obterVerificadorDeArqu
ivoDeletado(path), eagerComplete = true)
          .map(value => new ProducerRecord[String,
String]("contas", value))

.runWith(Producer.plainSink(kafkaProducerSettings))
    }
    mudancasNoDiretorio.runForeach {
      case (path, change) =>
        change match {
          case DirectoryChange.Creation =>
            obterVerificadorDeArquivoDeletado(path)

obterLeitorDeArquivo(path).merge(obterVerificadorDeArqu
```

```
ivoDeletado(path), eagerComplete = true)
                .map(value => new ProducerRecord[String,
String]("contas", value))

.runWith(Producer.plainSink(kafkaProducerSettings))
        }
    }
  }


  private def obterVerificadorDeArquivoDeletado(path:
Path): Source[Nothing, NotUsed] =
DirectoryChangesSource(path.getParent, 1 second, 8192)
    .collect {
      case (p, DirectoryChange.Deletion) if path == p
=>
        throw new FileNotFoundException(path.toString)
    }
    .recoverWithRetries(1, {
      case _: FileNotFoundException => Source.empty
    })

  private def obterLeitorDeArquivo(path: Path):
Source[String, NotUsed] = FileTailSource.lines(
    path = path,
    maxLineSize = 8192,
    pollingInterval = 250.millis
  )

}
```

Nele, realizamos os seguintes passos:

- Através do objeto `Directory`, realizamos a listagem de todos os arquivos contidos no diretório `./input_dir` utilizando o método `ls`.

Nessa listagem - ela própria também uma stream -, criamos uma stream para cada arquivo dentro do diretório.

- Analogamente, utilizando o objeto `DirectoryChangesSource`, criamos uma stream que ficará escutando mudanças no mesmo diretório e verificando se novos arquivos são colocados no diretório. Em caso afirmativo, realiza a mesma criação da stream que lista os arquivos do diretório, fazendo com que possamos "jogar" novos arquivos na pasta, que são imediatamente integrados para o Kafka sem a necessidade de reiniciar a aplicação.
- Para a criação das streams que ficam escutando os arquivos, criamos dois métodos privados utilitários, `obterLeitorDeArquivo` e `obterVerificadorDeArquivoDeletado`. O leitor de arquivos é implementado pelo objeto `FileTailSource`, que serve de nascente para a nossa stream. Essa nascente implementa um `tail` no arquivo, não só lendo todo o seu conteúdo na inicialização como também novas linhas que sejam inseridas no arquivo. O verificador de arquivo deletado implementa outra stream utilizando o mesmo objeto que escuta mudanças de diretório, porém, neste caso, para escutar mudanças no arquivo. Ela será responsável por observar se o arquivo foi

deletado, fazendo com que a stream seja encerrada nesse caso, em vez de continuar "ligada" esperando dados de um arquivo que já não existe mais. As duas streams são combinadas através do método `merge`.

- Por fim, temos o envio para o Kafka. Para isso, utilizamos o objeto `Producer` do conector Kafka do Alpakka. Através do método `plainSink`, criamos uma foz que envia mensagens para o Kafka através das configurações fornecidas pela variável `kafkaProducerSettings`. Antes de enviar para a foz, realizamos um mapeamento através do comando `.map(value => new ProducerRecord[String, String]("contas", value))`. Com esse comando convertemos cada registro `csv` em uma mensagem Kafka, que contém a linha do arquivo como um objeto `String`. O objeto também recebe por parâmetro o nome do tópico para onde vamos enviar a mensagem, no caso, `contas`.

Por fim, vamos modificar o objeto `Boot`, fazendo com que nossas streams sejam criadas quando a aplicação for iniciada:

```
package com.casadocodigo


import akka.actor.ActorSystem
import com.casadocodigo.streams.ArquivoParaKafka
```

```
import com.casadocodigo.streams.PrimeiraStream

import scala.concurrent.ExecutionContextExecutor
import scala.language.postfixOps


object Boot extends App {

  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")
  implicit val ec: ExecutionContextExecutor =
system.dispatcher

  PrimeiraStream.primeiraStream()
  ArquivoParaKafka.iniciarStreams()


}
```

## Testando nossas streams

Vamos agora criar um arquivo inicial para o processamento das streams. Criaremos o diretório `input_dir` na raiz do projeto e o arquivo `test.csv` :

```
"Alexandre",39,112322211
"Lucebiane",35,112322212
"Maria Odete",18,123442245
"Jesus Cristo",33,43567845
"Teste",11,716253677
"Leonardo da Vinci",60,813762918
```

Antes de iniciarmos a aplicação, vamos iniciar a nossa stack do `docker-compose` através do comando:

```
docker-compose up -d
```

E utilizar um comando do Kafka para observarmos as mensagens chegando no tópico. Vamos executar o seguinte comando no terminal. Após a execução, nosso terminal ficará "preso" executando o programa `shell` do Kafka, e teremos que abrir outra janela para executar a aplicação:

```
 docker exec -t -i capitulo_7_8_kafka_1
/opt/kafka/bin/kafka-console-consumer.sh --topic contas
--bootstrap-server localhost:9092
```

Por fim, em outra janela de terminal, vamos executar o comando `sbt run` , iniciando a aplicação. Após alguns segundos, se observarmos a primeira janela do terminal, veremos o conteúdo do arquivo aparecendo no terminal na forma de mensagens sendo publicadas no Kafka:

```
   docker exec -t -i capitulo_7_8_kafka_1
/opt/kafka/bin/kafka-console-consumer.sh --topic contas
--bootstrap-server localhost:9092
"Alexandre",39,112322211
"Lucebiane",35,112322212
"Maria Odete",19,123442245
"Ana Carolina","18",98274391
"Eleutério","20",81274610
```

```
"Jesus Cristo",33,43567845
"Teste",11,716253677
"Leonardo da Vinci",60,813762918
"Pedro Álvares Cabral",42,98789234
"Dom Pedro I",50,90284092348
"Celina",42,98789234
"Alex",14,817236781236
"Dom Pedro II",50,90284092348
```

Vamos testar agora se, ao incluirmos uma nova linha, ela será entregue ao Kafka. Vamos incluir a seguinte linha ao arquivo `csv` :

```
"Celina",42,98789234
```

Após inserirmos uma nova linha, observaremos que, após alguns segundos, a linha é integrada ao Kafka:

```
   docker exec -t -i capitulo_7_8_kafka_1
/opt/kafka/bin/kafka-console-consumer.sh --topic contas
--bootstrap-server localhost:9092
"Alexandre",39,112322211
"Lucebiane",35,112322212
"Maria Odete",19,123442245
"Ana Carolina","18",98274391
"Eleutério","20",81274610
"Jesus Cristo",33,43567845
"Teste",11,716253677
"Leonardo da Vinci",60,813762918
"Pedro Álvares Cabral",42,98789234
"Dom Pedro I",50,90284092348
"Celina",42,98789234
"Alex",14,817236781236
"Dom Pedro II",50,90284092348
"Celina",42,98789234
```

Vamos testar se nossa aplicação está escutando novos arquivos. Criaremos um novo arquivo `csv` chamado `novocsv.csv` e incluiremos a seguinte linha:

```
"Alex",14,817236781236
```

Após darmos um *enter*, ao observar o fluxo de mensagens no Kafka, poderemos observar que a nova mensagem apareceu no terminal, indicando que uma nova stream foi adicionada para escutar o novo arquivo automaticamente:

```
    docker exec -t -i capitulo_7_8_kafka_1
/opt/kafka/bin/kafka-console-consumer.sh --topic contas
--bootstrap-server localhost:9092
"Alexandre",39,112322211
"Lucebiane",35,112322212
"Maria Odete",19,123442245
"Ana Carolina","18",98274391
"Eleutério","20",81274610
"Jesus Cristo",33,43567845
"Teste",11,716253677
"Leonardo da Vinci",60,813762918
"Pedro Álvares Cabral",42,98789234
"Dom Pedro I",50,90284092348
"Celina",42,98789234
"Alex",14,817236781236
"Dom Pedro II",50,90284092348
"Celina",42,98789234
"Alex",14,817236781236
```

## Testando a detecção de arquivos por parte das streams

Por fim, vamos testar a deleção. Vamos deletar o arquivo `novocsv.csv` e, em seguida, inserir a seguinte linha no arquivo `test.csv` . O objetivo deste passo é verificar que a remoção de um arquivo não afeta em nada o processamento dos outros arquivos, tendo o término de seu processamento sendo executado de maneira limpa e sem grandes impactos. Incluímos a seguinte linha no arquivo, conforme dito anteriormente:

```
"Dom Pedro II",50,90284092348
```

E, como esperado, temos a nova linha aparecendo no nosso Kafka:

```
   docker exec -t -i capitulo_7_8_kafka_1
/opt/kafka/bin/kafka-console-consumer.sh --topic contas
--bootstrap-server localhost:9092
"Alexandre",39,112322211
"Lucebiane",35,112322212
"Maria Odete",19,123442245
"Ana Carolina","18",98274391
"Eleutério","20",81274610
"Jesus Cristo",33,43567845
"Teste",11,716253677
"Leonardo da Vinci",60,813762918
"Pedro Álvares Cabral",42,98789234
"Dom Pedro I",50,90284092348
"Celina",42,98789234
"Alex",14,817236781236
```

```
"Dom Pedro II",50,90284092348
"Celina",42,98789234
"Alex",14,817236781236
"Dom Pedro II",50,90284092348
```

Sucesso! Temos nossa primeira integração!

Vamos para o nosso próximo exemplo. Vamos fazer uma stream que vai consumir os dados desse tópico que acabamos de criar e inseri-los em uma base de dados utilizando o `Slick` , que vimos no capítulo anterior.

## 8.2 Integrando o Kafka com uma base de dados relacional

Agora, vamos criar outra stream. Nossa segunda stream vai ler os dados publicados no tópico Kafka da primeira stream, transformá-los em um objeto e inserir esse objeto em uma tabela de banco de dados. Para isso, utilizaremos outros dois componentes do Alpakka, o de `parsing` de arquivos `csv` e o que faz conexão com bancos de dados através do Slick, que vimos no capítulo anterior.

Para começar, vamos acrescentar arquivos de configuração, como os que vimos anteriormente, para encapsular nossas configurações de conexão

com o banco de dados. Para começar, vamos criar o arquivo `application.conf` , com as configurações padrões:

```
include "application.base.conf"
```

E o arquivo `application.base.conf` , onde criamos a configuração com a base de dados:

```
bootstrapServers = "localhost:9092"

# Datasources
slick-postgres {
  profile = "slick.jdbc.PostgresProfile$"
  db {
    dataSourceClass = "slick.jdbc.DriverDataSource"
    properties = {
      driver = "org.postgresql.Driver"
      url =
"jdbc:postgresql://127.0.0.1:5432/ecommerce"
      url = ${?DB_URL}
      user = postgres
      user = ${?DB_USER}
      password = "teste"
      password = ${?DB_PASS}
    }
  }
}
```

O leitor pode notar que utilizamos um formato um pouco diferente do que utilizamos no capítulo anterior. Esse formato permite que criemos outro objeto da biblioteca do Slick, o `SlickSession`, que é utilizado no padrão utilizado pelo conector do Alpakka, conforme veremos a seguir.

Nos arquivos de configuração, incluímos também uma propriedade chamada `bootstrapServers`. Essa propriedade nos será útil para que não precisemos repetir o endereço do Kafka em ambas as streams. Assim, vamos modificar o arquivo `ArquivoParaKafka`, fazendo com que ele utilize a configuração para obter o endereço do servidor em vez de colocá-lo como *hard-coded* diretamente, como vemos no fragmento a seguir:

```
...
  private val kafkaProducerSettings =
    ProducerSettings.create(system, new
StringSerializer(), new StringSerializer())

.withBootstrapServers(config.getString("bootstrapServer
s"))
...
```

Vamos também atualizar o `build.sbt`, incluindo as dependências dos conectores que vamos utilizar:

```
name := "akka-streams"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"
val slickVersion = "3.3.3"

mainClass in Compile := Some("com.casadocodigo.Boot")

//Akka e outras dependencias
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test
)

// dependencias de logging
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)

// dependencias do Alpakka
libraryDependencies ++= Seq(
  "com.lightbend.akka" %% "akka-stream-alpakka-file" %
"3.0.3",
  "com.typesafe.akka" %% "akka-stream-kafka" % "2.1.1",
  "com.lightbend.akka" %% "akka-stream-alpakka-slick" %
"3.0.3",
  "com.lightbend.akka" %% "akka-stream-alpakka-csv" %
"3.0.3"
)
```

```
// dependencias de banco
libraryDependencies ++= Seq(
  "org.postgresql" % "postgresql" % "42.2.23"
)
```

## Transformando os dados para persistência no banco

Agora, vamos criar o objeto que cria a nossa segunda stream. No nosso objeto, criamos um mapeamento do Slick a fim de fazer com que ele crie a nossa tabela durante a inicialização. Na stream, utilizaremos o método `Consumer.plainSource` para consumir os dados do Kafka. Essa nascente cria objetos do tipo `ConsumerRecord` , que possuem o conteúdo da mensagem dentro de uma propriedade chamada `value` .

A seguir, vamos transformar a mensagem de `csv` para o objeto de banco. Isso é feito através dos objetos `CsvParsing e CsvToMap` . O `CsvParsing` é responsável por extrair os valores da linha. Esse objeto recebe `ByteStrings` como entrada, por isso transforma os dados de `String` para `ByteString` antes de repassá-los para o objeto.

A seguir, o objeto `CsvToMap` mapeia os dados extraídos para uma estrutura de dados do tipo mapa, por meio da qual conseguimos obter os dados

através das chaves. Com o mapeamento `.map(registro => Conta(registro("nome"), registro("idade").toInt, registro("documento").toLong))`, que fazemos a seguir, criamos objetos do tipo `Conta` a partir dos dados do mapa.

Por fim, utilizando o método `Slick.sink` do conector de Slick do Alpakka, criamos uma foz na qual os dados dos objetos do tipo `Conta` criados são utilizados em inserções no banco de dados, finalizando a nossa stream.

O código a seguir cria a stream, seguindo os passos que falamos anteriormente:

```
package com.casadocodigo.streams

import akka.kafka.{ConsumerSettings, Subscriptions}
import akka.kafka.scaladsl.Consumer
import akka.stream.alpakka.csv.scaladsl.{CsvParsing,
CsvToMap}
import akka.stream.alpakka.slick.scaladsl.Slick
import akka.util.ByteString
import org.apache.kafka.clients.consumer.ConsumerConfig
import
org.apache.kafka.common.serialization.StringDeserialize
r
import com.casadocodigo.Boot.{config, system, session}
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import java.nio.charset.StandardCharsets
```

```
object KafkaParaBanco {

  case class Conta(nome: String, idade: Int, documento:
Long)

  class ContaSchema(tag: Tag) extends Table[Conta](tag,
"conta") {
    def nome = column[String]("nome")

    def idade = column[Int]("idade")

    def documento = column[Long]("documento")

    def * = (nome, idade, documento) <> (Conta.tupled,
Conta.unapply)
  }

  private val db = session.db
  private val tabela = TableQuery[ContaSchema]
  db.run(DBIO.seq(
    tabela.schema.createIfNotExists
  ))

  private val consumerSettings =
    ConsumerSettings(system, new StringDeserializer,
new StringDeserializer)

.withBootstrapServers(config.getString("bootstrapServer
s"))
      .withGroupId("grupo")

.withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest")

.withProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
"true")
```

```
.withProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CO
NFIG, "5000")

  def iniciarStreams(): Unit = {
    Consumer
      .plainSource(
        consumerSettings,
        Subscriptions.topics("contas")
      ).map(registro => {
      val linhaComQuebraDelinha = registro.value() +
"\n"
      ByteString(linhaComQuebraDelinha)
    }
    )
      .via(CsvParsing.lineScanner())

.via(CsvToMap.withHeadersAsStrings(StandardCharsets.UTF
_8, "nome", "idade", "documento"))
      .map(registro => Conta(registro("nome"),
registro("idade").toInt, registro("documento").toLong))
      .runWith(
        Slick.sink(conta => sqlu"INSERT INTO conta
VALUES(${conta.nome}, ${conta.idade},
${conta.documento})")
      )
  }

}
```

A seguir, vamos modificar o arquivo `Boot` a fim de incluir os objetos necessários para o acesso à base de dados. Nele, nós criaremos o objeto `SlickSession`, que citamos há pouco, informando que ele deve encerrar a conexão com o banco

quando o sistema de atores for encerrado, caso nossa aplicação seja encerrada. Também incluiremos o objeto `KafkaParaBanco` e faremos com que a nova stream seja inicializada quando a aplicação for inicializada. Por fim, inicializaremos também o objeto `config`, utilizado pelas streams, para obter o endereço do Kafka:

```
package com.casadocodigo


import akka.actor.ActorSystem
import akka.stream.alpakka.slick.scaladsl.SlickSession
import com.casadocodigo.streams.{ArquivoParaKafka,
KafkaParaBanco, PrimeiraStream}
import com.typesafe.config.{Config, ConfigFactory}

import scala.concurrent.ExecutionContextExecutor
import scala.language.postfixOps


object Boot extends App {

  implicit val config: Config =
ConfigFactory.load(Option(
    System.getenv("ENVIRONMENT"))

.getOrElse(Option(System.getProperty("ENVIRONMENT"))
      .getOrElse("application")))
  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")
  implicit val ec: ExecutionContextExecutor =
system.dispatcher
  implicit val session: SlickSession =
SlickSession.forConfig("slick-postgres")
```

```
  system.registerOnTermination(() => session.close())

  PrimeiraStream.primeiraStream()
  ArquivoParaKafka.iniciarStreams()
  KafkaParaBanco.iniciarStreams()

}
```

Continuando, vamos modificar o arquivo `docker-compose.yml` . Nessa modificação, incluiremos o banco de dados, fazendo com que ele suba em conjunto com o nosso Kafka:

```
version: '3.9'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
    ports:
      - 2181:2181
  kafka:
    image: wurstmeister/kafka:2.12-2.5.0
    ports:
      - 9092:9092
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_CREATE_TOPICS: "contas:1:1"
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
  db:
    image: postgres
    ports:
      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: teste
      POSTGRES_DB: ecommerce
```

Finalizando a codificação, vamos criar um `shell script` chamado `run_local.sh` . Nesse arquivo, vamos definir que nossos recursos sejam recriados a fim de sempre termos um ambiente inicial estável e rodar a nossa aplicação a seguir, iniciando todas as streams:

```
#!/usr/bin/env bash

docker-compose down
docker-compose up -d

sleep 10

sbt run
```

Vamos testar as streams. Vamos deixar dentro do diretório `input_dir` apenas um arquivo `csv` chamado `test.csv` com o seguinte conteúdo:

```
"Alexandre",39,112322211
"Lucebiane",35,112322212
"Maria Odete",19,123442245
"Ana Carolina","18",98274391
"Eleutério","20",81274610
"Jesus Cristo",33,43567845
"Teste",11,716253677
"Leonardo da Vinci",60,813762918
"Pedro Álvares Cabral",42,98789234
"Dom Pedro I",50,90284092348
```

A seguir, vamos executar o `shell script` , através do comando:

```
./run_local.sh
```

A princípio, não veremos nada no console além dos textos da nossa primeira stream de exemplo, como podemos ver na imagem a seguir:

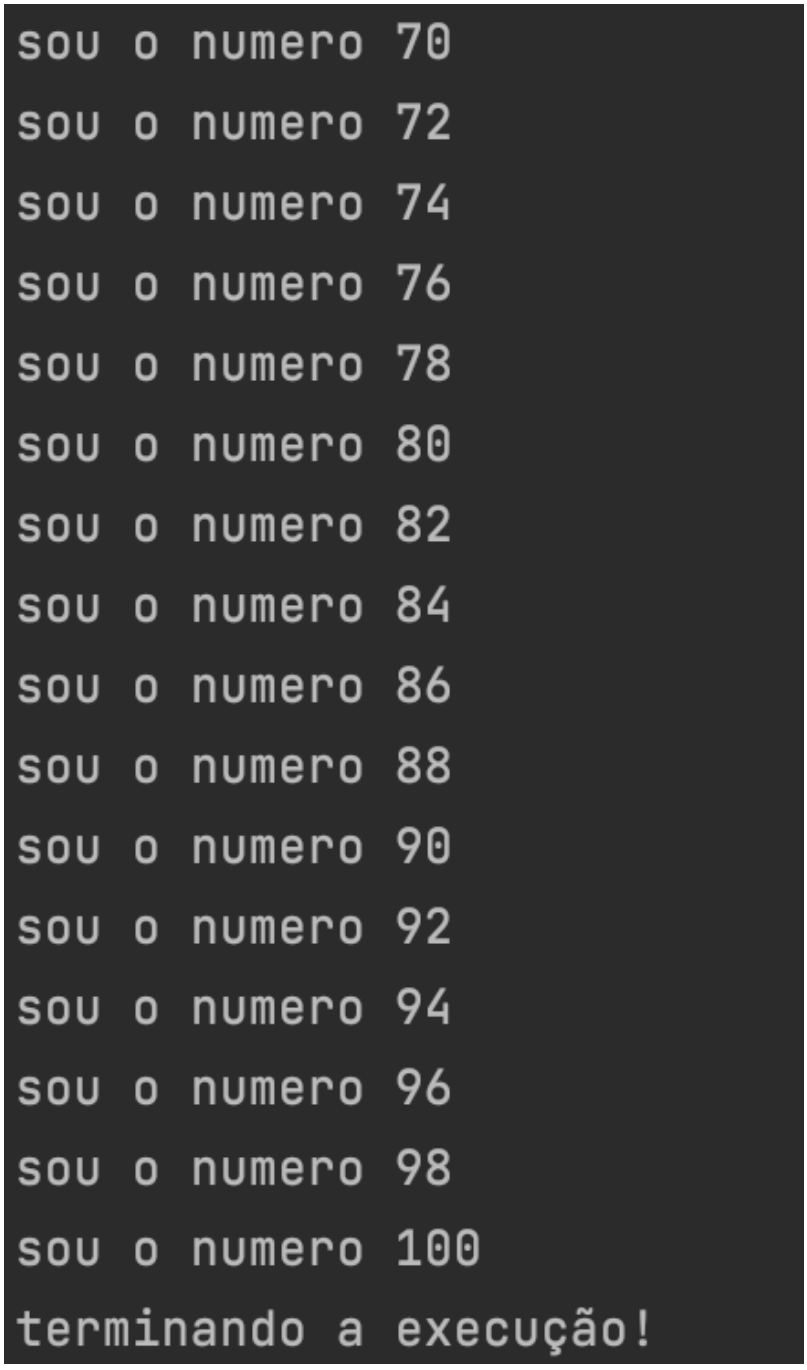



Figura 8.1: Fragmento do console do final da execução no terminal.

Vamos verificar que está tudo certo? Vamos nos conectar ao banco de dados utilizando o seu programa favorito. Usarmos as mesmas configurações do capítulo anterior:

- host: localhost
- port: 5432
- database: ecommerce
- user: postgres
- password: teste

Após nos conectarmos, navegamos até o *schema* public do banco e verificamos que a tabela não só foi criada com sucesso, como os dados do arquivo foram inseridos com sucesso:

| | nome<br>character varying | idade<br>integer | documento<br>bigint |
|---|---|---|---|
| 1 | Alexandre | 39 | 112322211 |
| 2 | Lucebiane | 35 | 112322212 |
| 3 | Maria Odete | 19 | 123442245 |
| 4 | Ana Carolina | 18 | 98274391 |
| 5 | Eleutério | 20 | 81274610 |
| 6 | Jesus Cristo | 33 | 43567845 |
| 7 | Teste | 11 | 716253677 |
| 8 | Leonardo da Vinci | 60 | 813762918 |
| 9 | Pedro Álvares Cabral | 42 | 98789234 |
| 10 | Dom Pedro I | 50 | 90284092348 |

Figura 8.2: Dados do csv inseridos no banco de dados.

Sucesso! Temos nosso pipeline de leitura de dados vindos de um arquivo, passando pelo Kafka e por fim sendo persistidos em uma base de dados.

Vamos agora imaginar um cenário em que também precisamos cadastrar os nossos clientes em APIs,

além de persistir no banco de dados. Como faremos isso? É o que veremos a seguir na nossa próxima stream!

## 8.3 Integrando o Kafka com APIs REST

Vamos supor que, além criar a conta na base de dados, nosso cadastro de contas também necessita chamar as APIs das áreas de crédito e fiscal a fim de cadastrar a conta nos sistemas necessários para que ela se torne operacional. Todas essas chamadas e a persistência podem ser feitas em paralelo sem a necessidade de que elas tenham uma ordem de execução.

Assim, podemos construir uma nova stream que também obtém os registros do tópico Kafka, processando em paralelo com a persistência do banco. Por fim, as chamadas às APIs serão feitas em paralelo na stream para paralelizar todos os passos.

### Configurando o ambiente

Vamos começar incluindo as dependências do Akka HTTP. O projeto Alpakka não possui conectores próprios para operar com HTTP, recomendando utilizar o Akka HTTP diretamente. Também

incluiremos a biblioteca `spray-json` , que utilizaremos para serializar as requisições das chamadas HTTP:

```
name := "akka-streams"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"
val slickVersion = "3.3.3"
val akkaHttpVersion = "10.2.6"

mainClass in Compile := Some("com.casadocodigo.Boot")

//Akka e outras dependencias
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-http" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-spray-json" %
akkaHttpVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test
)

// dependencias de logging
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)

// dependencias do Alpakka
libraryDependencies ++= Seq(
```

```
  "com.lightbend.akka" %% "akka-stream-alpakka-file" %
"3.0.3",
  "com.typesafe.akka" %% "akka-stream-kafka" % "2.1.1",
  "com.lightbend.akka" %% "akka-stream-alpakka-slick" %
"3.0.3",
  "com.lightbend.akka" %% "akka-stream-alpakka-csv" %
"3.0.3"
)

// dependencias de banco
libraryDependencies ++= Seq(
  "org.postgresql" % "postgresql" % "42.2.23"
)
```

Vamos modificar o arquivo `application.base.conf` incluindo a propriedade `url`, que contém o `host` das chamadas HTTP que vamos realizar:

```
bootstrapServers = "localhost:9092"
url = "localhost"


# Datasources
slick-postgres {
  profile = "slick.jdbc.PostgresProfile$"
  db {
    dataSourceClass = "slick.jdbc.DriverDataSource"
    properties = {
      driver = "org.postgresql.Driver"
      url =
"jdbc:postgresql://127.0.0.1:5432/ecommerce"
      url = ${?DB_URL}
      user = postgres
      user = ${?DB_USER}
      password = "teste"
```

```
      password = ${?DB_PASS}
    }
  }
}
```

Nas nossas chamadas HTTP, enviaremos a nossa classe `Conta` como um JSON na requisição, por isso vamos criar também um serializador chamado `SerializadorJSON` com o seguinte código:

```
package com.casadocodigo.streams

import
akka.http.scaladsl.marshallers.sprayjson.SprayJsonSuppo
rt
import com.casadocodigo.streams.KafkaParaBanco.Conta
import spray.json.DefaultJsonProtocol

trait SerializadorJSON extends SprayJsonSupport with
DefaultJsonProtocol {

  implicit val contaFormat = jsonFormat3(Conta)

}
```

## Criando a stream de integração com as APIs

Vamos criar o objeto `KafkaParaAPIs` no qual criaremos nossa nova stream. Diversos novos conceitos são apresentados neste objeto:

- Primeiramente, definimos um `Supervision.Decider` com a estratégia de *restart* para todos os tipos de exceções. Vamos

utilizar isso para demonstrar como streams também permitem a configuração de supervisores para tratamento de erros, da mesma forma que vimos com atores.

- A seguir, criamos as configurações de acesso ao Kafka, como vimos na stream anterior.
- As principais novidades estão na variável `grafo`. Dentro da variável, que criamos a partir do método `Flow.fromGraph(`, criamos toda a sequência de passos que compõe a parte da stream responsável pela comunicação com as APIs, conforme descreveremos a seguir:
    - Primeiramente, criamos um difusor chamado `difusorMensagemKafka`. Esse componente recebe mensagens de entrada do fluxo e cria duas cópias da mensagem que envia para suas duas portas de saída.
    - Para o outro extremo da execução, criamos um juntor chamado `juntorMensagemsAPIs`. Esse juntor recebe dois resultados que são as chamadas às APIs. Quando os resultados são recebidos, ele os une em uma *tupla* - estrutura de dados posicional parecida com uma lista indexada -, passando adiante a tupla com a união dos resultados.
    - Nas variáveis `enviaParaFiscal` e `enviaParaCredito`, efetuamos as chamadas HTTP. O código é bastante

parecido com o que já fizemos no capítulo 5, por isso não necessita de maiores explicações.

- Assim, temos a montagem da stream propriamente dita, unindo o difusor, as chamadas e o juntor. Isso é feito através do operador implícito `~>`, que importamos através da linha `import GraphDSL.Implicits._`. Repare que, para paralelizar os passos, tudo o que fizemos foi colocar as linhas uma em baixo da outra, permitindo que implementemos o fluxo de maneira quase visual, como podemos ver no fragmento:

```
  ...
  difusorMensagemKafka ~> enviaParaFiscal ~>
juntorMensagemsAPIs.in0
  difusorMensagemKafka ~> enviaParaCredito ~>
juntorMensagemsAPIs.in1
  ...
```

- Por fim, criamos um objeto do tipo `FlowShape` no qual passamos o difusor e o juntor como entrada e saída do fluxo, concluindo a construção desse trecho da stream. Esse trecho, que contém diversos passos conectados formando um fluxo, conforme dito anteriormente, é definido como

uma estrutura de dados do tipo grafo dentro do Akka Streams.

- A seguir, criamos a stream propriamente dita. O início da stream consiste em instanciar o consumidor do Kafka e passar a mensagem do formato csv para um objeto do tipo `Conta`, como já fizemos antes. Para utilizar o trecho que criamos anteriormente, utilizamos o método `via`, passando por parâmetro a variável `grafo`.
- Por fim, verificamos no resultado da execução se tivemos sucesso nas operações (HTTP 200 OK) ou não. Dependendo dos resultados, logamos o sucesso ou fracasso da operação.

O código a seguir cria tudo o que descrevemos anteriormente:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.http.scaladsl.Http
import akka.http.scaladsl.model.{ContentTypes,
HttpEntity, HttpMethods, HttpRequest}
import akka.kafka.scaladsl.Consumer
import akka.kafka.{ConsumerSettings, Subscriptions}
import akka.stream.{ActorAttributes, FlowShape,
Supervision}
import akka.stream.alpakka.csv.scaladsl.{CsvParsing,
CsvToMap}
import akka.stream.scaladsl.{Broadcast, Flow, GraphDSL,
Sink, Source, Zip}
```

```
import akka.util.ByteString
import com.casadocodigo.Boot.{config, system}
import com.casadocodigo.streams.KafkaParaBanco.Conta
import org.apache.kafka.clients.consumer.ConsumerConfig
import
org.apache.kafka.common.serialization.StringDeserialize
r
import spray.json._

import java.nio.charset.StandardCharsets
import scala.language.postfixOps
import scala.util.Success

object KafkaParaAPIs extends SerializadorJSON {

  private val decider: Supervision.Decider = _ =>
Supervision.Restart

  private val consumerSettings =
    ConsumerSettings(system, new StringDeserializer,
new StringDeserializer)

.withBootstrapServers(config.getString("bootstrapServer
s"))
      .withGroupId("grupoAPIs")

.withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest")

.withProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
"true")

.withProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CO
NFIG, "5000")

  def iniciarStreams(): Unit = {
```

```
    val grafo = Flow.fromGraph(GraphDSL.create() {
implicit builder =>

      import GraphDSL.Implicits._

      val difusorMensagemKafka =
builder.add(Broadcast[Conta](2))
      val juntorMensagemsAPIs = builder.add(Zip[Int,
Int])

      val enviaParaFiscal = Flow[Conta].map(
        msg => {
          println(s"processando a conta
${msg.toJson.compactPrint} para o fiscal")
          (HttpRequest(method = HttpMethods.POST,
            uri =
s"http://${config.getString("url")}:3000/fiscal",
            entity =
HttpEntity(ContentTypes.`application/json`,
msg.toJson.compactPrint)), NotUsed)
        }
      )
        .via(Http().superPool[NotUsed]())
        .map(
          resposta => {
            println(s"resposta do fiscal: $resposta")
            resposta._1.get.status.intValue()
          }
        )

      val enviaParaCredito = Flow[Conta].map(
        msg => {
          println(s"processando a conta
${msg.toJson.compactPrint} para o credito")
          (HttpRequest(method = HttpMethods.POST,
            uri =
s"http://${config.getString("url")}:3000/credito",
```

```
            entity =
HttpEntity(ContentTypes.`application/json`,
msg.toJson.compactPrint)), NotUsed)
        }
      )
        .via(Http().superPool[NotUsed]())
        .map(
          resposta => {
            println(s"resposta do credito: $resposta")
            resposta._1.get.status.intValue()
          }
        )

      difusorMensagemKafka ~> enviaParaFiscal ~>
juntorMensagemsAPIs.in0
      difusorMensagemKafka ~> enviaParaCredito ~>
juntorMensagemsAPIs.in1

      FlowShape(difusorMensagemKafka.in,
juntorMensagemsAPIs.out)

    })
    Consumer
      .plainSource(
        consumerSettings,
        Subscriptions.topics("contas")
      ).map(registro => {
      val linhaComQuebraDelinha = registro.value() +
"\n"
      ByteString(linhaComQuebraDelinha)
    }
    )
      .via(CsvParsing.lineScanner())

.via(CsvToMap.withHeadersAsStrings(StandardCharsets.UTF
_8, "nome", "idade", "documento"))
      .map(registro => Conta(registro("nome"),
```

```
registro("idade").toInt, registro("documento").toLong))
      .map(conta => {
        println(s"processando a conta $conta")
        conta
      })
      .via(grafo)
      .map(
        respostas =>
          if (respostas._1 != 200 || respostas._2 !=
200) {
            println("Problema para acessar o fiscal ou
o credito! Favor checar os logs.")
          } else {
            println("Conta processada com sucesso!")
          }
      )

.withAttributes(ActorAttributes.supervisionStrategy(dec
ider))
      .runWith(Sink.ignore)
  }

}
```

E por fim, atualizamos o objeto `Boot`, adicionando o objeto `KafkaParaAPIs` à lista de streams que inicializamos quando a aplicação é iniciada:

```
package com.casadocodigo


import akka.actor.ActorSystem
import akka.stream.alpakka.slick.scaladsl.SlickSession
import com.casadocodigo.streams.{ArquivoParaKafka,
KafkaParaAPIs, KafkaParaBanco, PrimeiraStream}
import com.typesafe.config.{Config, ConfigFactory}
```

```
import scala.concurrent.ExecutionContextExecutor
import scala.language.postfixOps


object Boot extends App {

  implicit val config: Config =
ConfigFactory.load(Option(
    System.getenv("ENVIRONMENT"))

.getOrElse(Option(System.getProperty("ENVIRONMENT"))
      .getOrElse("application")))
  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")
  implicit val ec: ExecutionContextExecutor =
system.dispatcher
  implicit val session: SlickSession =
SlickSession.forConfig("slick-postgres")
  system.registerOnTermination(() => session.close())

  PrimeiraStream.primeiraStream()
  ArquivoParaKafka.iniciarStreams()
  KafkaParaBanco.iniciarStreams()
  KafkaParaAPIs.iniciarStreams()

}
```

## Criando os mocks das APIs

Antes de começarmos a testar a stream, vamos apenas criar os nossos mocks das APIs no `Mockoon`. Primeiramente, vamos criar o mock para o endpoint da área Fiscal, da seguinte forma:

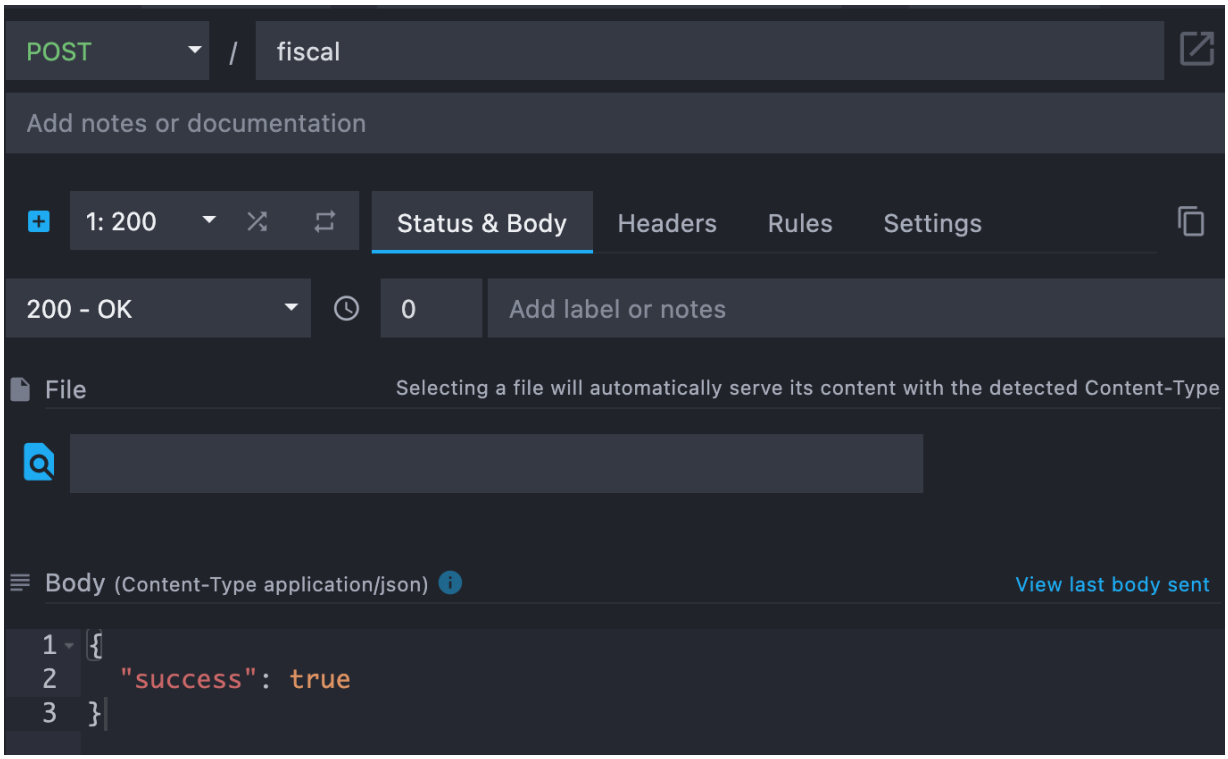



Figura 8.3: Mock da API da área fiscal.

E o mock para o endpoint da área de crédito:

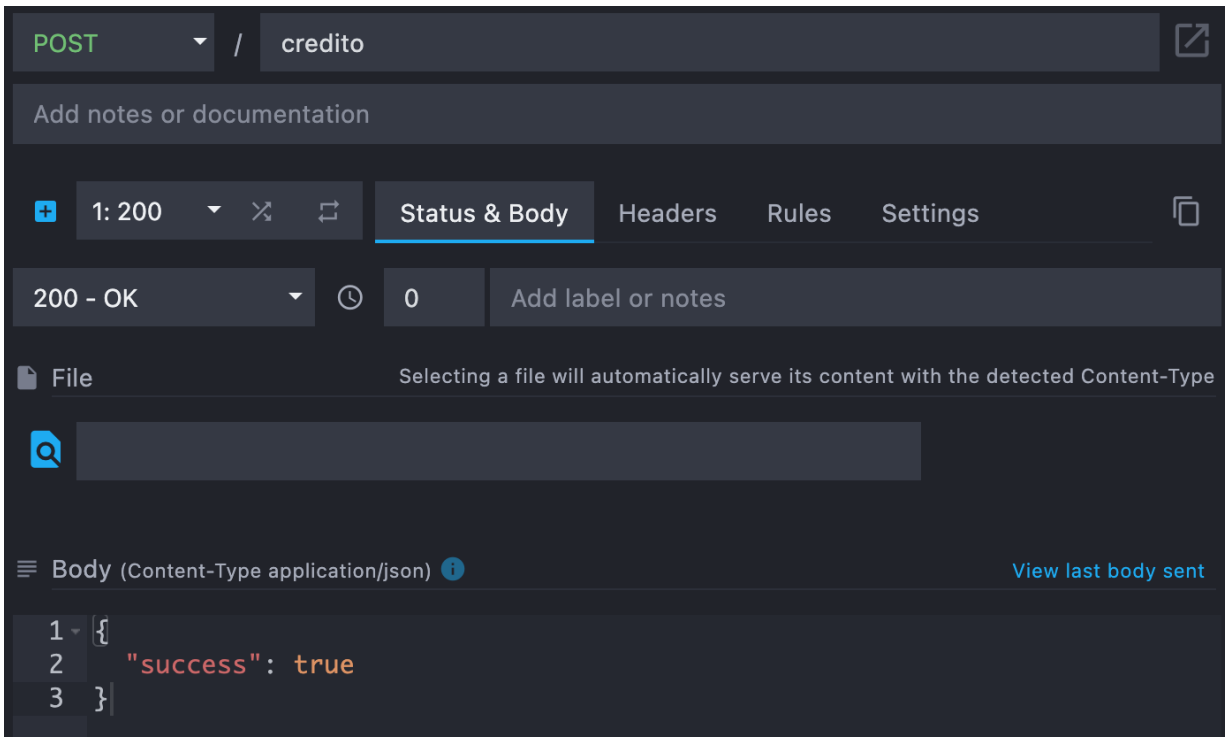



Figura 8.4: Mock da API da área de crédito.

Vamos agora testar o nosso código!

Como primeiro teste, vamos iniciar o `Mockoon` e a seguir executar o `shell script run_local.sh`. Após alguns segundos, podemos ver mensagens de retorno das chamadas HTTP, bem como as mensagens finais de sucesso `Conta processada com sucesso!`, como podemos ver no fragmento a seguir:

```
...
Conta processada com sucesso!
processando a conta Conta(Eleutério,20,81274610)
processando a conta
{"documento":81274610,"idade":20,"nome":"Eleutério"}
```

```
para o fiscal
processando a conta
{"documento":81274610,"idade":20,"nome":"Eleutério"}
para o credito
resposta do fiscal: HttpResponse(200 OK,List(Date: Wed,
01 Sep 2021 00:52:09 GMT, Connection: keep-alive, Keep-
Alive: timeout=5),HttpEntity.Strict(application/json,21
bytes total),HttpProtocol(HTTP/1.1))
resposta do credito: HttpResponse(200 OK,List(Date:
Wed, 01 Sep 2021 00:52:09 GMT, Connection: keep-alive,
Keep-Alive:
timeout=5),HttpEntity.Strict(application/json,21 bytes
total),HttpProtocol(HTTP/1.1))
Conta processada com sucesso!
processando a conta Conta(Jesus Cristo,33,43567845)
processando a conta
{"documento":43567845,"idade":33,"nome":"Jesus Cristo"}
para o fiscal
processando a conta
{"documento":43567845,"idade":33,"nome":"Jesus Cristo"}
para o credito
resposta do fiscal: HttpResponse(200 OK,List(Date: Wed,
01 Sep 2021 00:52:09 GMT, Connection: keep-alive, Keep-
Alive: timeout=5),HttpEntity.Strict(application/json,21
bytes total),HttpProtocol(HTTP/1.1))
resposta do credito: HttpResponse(200 OK,List(Date:
Wed, 01 Sep 2021 00:52:09 GMT, Connection: keep-alive,
Keep-Alive:
timeout=5),HttpEntity.Strict(application/json,21 bytes
total),HttpProtocol(HTTP/1.1))
Conta processada com sucesso!
...
```

## Testando erros de chamadas às APIs

Vamos agora fazer um segundo teste. Vamos modificar o mock da API da área de crédito para retornar um resultado HTTP 400 em vez de 200:

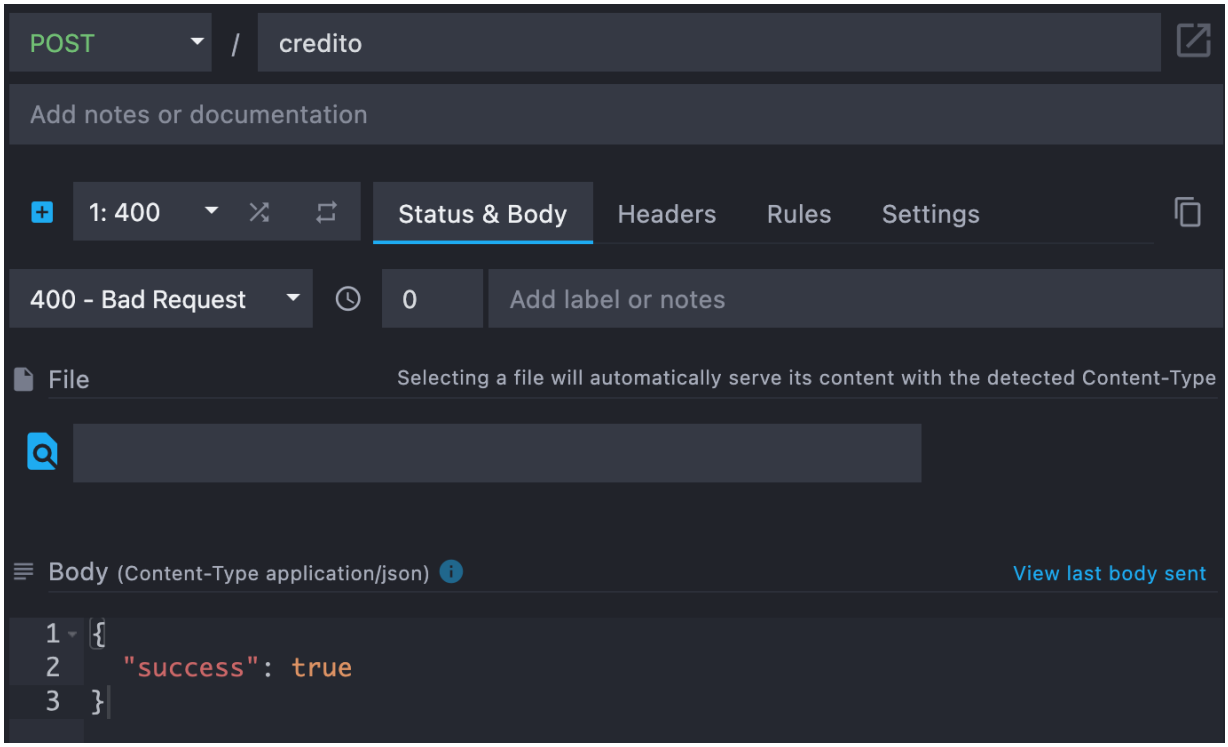



Figura 8.5: Mock da API da área de crédito (HTTP 400).

Executando novamente o `shell script`, podemos ver que as mensagens mudaram, refletindo os problemas no retorno de uma das APIs:

```
...
Problema para acessar o fiscal ou o credito! Favor
checar os logs.
processando a conta Conta(Pedro Álvares
Cabral,42,98789234)
processando a conta
```

```
{"documento":98789234,"idade":42,"nome":"Pedro Álvares
Cabral"} para o fiscal
processando a conta
{"documento":98789234,"idade":42,"nome":"Pedro Álvares
Cabral"} para o credito
resposta do fiscal: HttpResponse(200 OK,List(Date: Wed,
01 Sep 2021 01:00:47 GMT, Connection: keep-alive, Keep-
Alive: timeout=5),HttpEntity.Strict(application/json,21
bytes total),HttpProtocol(HTTP/1.1))
resposta do credito: HttpResponse(400 Bad
Request,List(Date: Wed, 01 Sep 2021 01:00:47 GMT,
Connection: keep-alive, Keep-Alive:
timeout=5),HttpEntity.Strict(application/json,21 bytes
total),HttpProtocol(HTTP/1.1))
Problema para acessar o fiscal ou o credito! Favor
checar os logs.
...
```

Finalmente, vamos fazer um último teste no qual executaremos a stream com o `Mockoon` parado. O que deve acontecer nesse caso é que veremos as mensagens sendo processadas até o ponto em que as chamadas HTTP são feitas, partindo em seguida para a próxima mensagem.

A razão para isso é porque definimos a estratégia de *restart*, que descarta a mensagem e reinicializa a stream. No nosso exemplo, utilizamos o Kafka com `commit` automático, por isso, assim que lemos uma mensagem do Kafka, ela já é marcada como processada e o Kafka aguarda a leitura da próxima mensagem, independentemente do resultado do processamento.

No nosso exemplo, a perda de dados não é um problema, mas em um caso real, podemos modificar a nascente para realizar `commits` manuais no processamento das mensagens do Kafka, somente passando para a próxima mensagem caso as chamadas HTTP retornem com sucesso. Deixo para você o desafio de modificar a stream para utilizar `commit` manual. O fragmento a seguir mostra a execução descrita acima na prática:

```
processando a conta Conta(Alexandre,39,112322211)
processando a conta
{"documento":112322211,"idade":39,"nome":"Alexandre"}
para o fiscal
processando a conta
{"documento":112322211,"idade":39,"nome":"Alexandre"}
para o credito
processando a conta Conta(Lucebiane,35,112322212)
processando a conta
{"documento":112322212,"idade":35,"nome":"Lucebiane"}
para o fiscal
processando a conta
{"documento":112322212,"idade":35,"nome":"Lucebiane"}
para o credito
processando a conta Conta(Maria Odete,19,123442245)
processando a conta
{"documento":123442245,"idade":19,"nome":"Maria Odete"}
para o fiscal
processando a conta
{"documento":123442245,"idade":19,"nome":"Maria Odete"}
para o credito
processando a conta Conta(Ana Carolina,18,98274391)
processando a conta
{"documento":98274391,"idade":18,"nome":"Ana Carolina"}
```

```
para o fiscal
processando a conta
{"documento":98274391,"idade":18,"nome":"Ana Carolina"}
para o credito
```

E assim concluímos o nosso estudo do Akka Streams. Como podemos ver, ele é um recurso bastante poderoso que nos permite criar integrações bastante robustas, porém de maneira simples usufruindo dos benefícios do Akka. A seguir, vamos aprender sobre testes automatizados e ver como podemos testar nossas aplicações em Akka e Akka Streams.

## Capítulo 9

# Testes automatizados com Akka e Akka Streams

Continuando nossos estudos, vamos agora aprender como criar testes automatizados de nossas aplicações com Akka e Akka Streams.

Testes automatizados são muito importantes no desenvolvimento de software. Através de testes, obtemos de forma automática diversos benefícios:

- Validar se refatorações no código não quebraram o código;
- Validar se novos desenvolvimentos não vão introduzir erros ou comportamentos indesejados no código já existente;
- Validar se a interface pública do código está adequada. Testes sempre invocam a camada "pública" do código, ou seja, as camadas que podem ser chamadas externamente. Desse modo, se em um teste temos grande dificuldade para conseguir montar o cenário de testes, isso pode significar que a camada pública do código não está bem definida.

Assim, testes são uma parte importante da vida de todo desenvolvedor e desenvolvedora de software.

São tão importantes que toda uma metodologia de desenvolvimento foi criada em torno deles, o famoso *TDD* (*Test-Driven Development*). Na Casa do Código, há diversos livros abordando o TDD, se você ainda não é familiarizado com a metodologia, recomendo muito a leitura desses livros!

Agora, veremos como podemos realizar testes no Akka e no Akka Streams.

## 9.1 Testes no Akka

O Akka possui um kit de desenvolvimento de testes próprio, com classes, objetos e traits que nos permitem desenvolver esses testes. Além disso, também utilizaremos a biblioteca `scalatest`, que nos permite criar *cenários de testes*, ou seja, métodos em que implementamos diferentes tipos de entradas para a execução do nosso código, onde testamos como o nosso código se comportará mediante esses diferentes tipos de entradas.

Para os nossos testes, utilizaremos o código produzido nos capítulos 4, 7 e 8. Nele, temos streams e atores básicos que nos permitem aprender a base do kit de testes. Sugiro como um desafio para você implementar os testes da nossa API dos capítulos 5 e 6.

Assim sendo, vamos copiar o nosso projeto do capítulo 4 e modificar o arquivo `build.sbt` incluindo as dependências necessárias para criarmos os testes:

```
name := "akka-streams"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test,
  "org.scalatest" %% "scalatest" % "3.2.9" % Test
)

// dependencias de logging
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)
```

O kit de desenvolvimento de testes possui dois formatos de desenvolvimento:

- Testes síncronos.
- Testes assíncronos.

Os *testes síncronos* nos permitem realizar testes simples de atores. Nesse cenário, o kit de

desenvolvimento de testes não instancia um servidor de atores de fato, apenas instancia os atores e envia as mensagens como se fossem objetos Scala simples, sem de fato usar a infraestrutura do Akka. É útil para alguns testes simples, mas não permite que testemos recursos do servidor, como supervisores e agendadores.

Os *testes assíncronos* nos permitem testar atores de maneira completa com um servidor de atores. Nesse cenário, temos um ambiente mais próximo do ambiente real de execução da nossa aplicação com todos os recursos disponíveis para serem testados.

Vamos começar o nosso estudo aprendendo primeiramente como desenvolver testes síncronos.

Devido ao fato de o código dos projetos já ter sido apresentado nos capítulos anteriores, não vamos mostrá-lo neste capítulo. Vamos nos ater apenas ao código novo, ou seja, ao código dos testes. Caso necessite consultar o código dos projetos novamente, sugiro acessar os capítulos anteriores, ou o repositório Git do projeto: https://github.com/alexandreesl/livro-akka-streams

**Observação:** no caso dos testes de Akka Streams, que veremos na última seção, realizaremos algumas refatorações no código. Isso se deve ao nosso código não ter sido criado com a realização posterior de testes em mente. Conforme discutiremos com mais detalhes adiante, isso demonstra como o desenvolvimento orientado a testes é uma ferramenta bastante poderosa, pois teríamos detectado certas refatorações no código que o tornariam mais flexível se o tivéssemos criado junto aos testes.

## Testes síncronos

No nosso projeto, temos dois atores: o ator parável `StoppableActor`, que lança a exceção `ExcecaoDeFinalizacao` de acordo com o tipo de mensagem recebida. Para efetuar os testes desse

ator, criamos a classe `StoppableActorSpec` com o seguinte conteúdo:

```
package com.casadocodigo.actors.synchronous

import akka.actor.testkit.typed.CapturedLogEvent
import
akka.actor.testkit.typed.scaladsl.BehaviorTestKit
import com.casadocodigo.actors.StoppableActor
import com.casadocodigo.actors.StoppableActor.
{ExcecaoDeFinalizacao, Mensagem, MensagemFinalizar,
MensagemProcessamento}
import org.scalatest._
import flatspec._
import matchers._
import org.slf4j.event.Level


class StoppableActorSpec extends AnyFlatSpec with
should.Matchers {

  "O ator paravel" should "processar mensagens de
processamento" in {

    val testador: BehaviorTestKit[Mensagem] =
BehaviorTestKit[Mensagem](StoppableActor.behavior())

    val dado = "Testando!"

    testador.run(MensagemProcessamento(dado))

    testador.logEntries() shouldBe
Seq(CapturedLogEvent(Level.INFO, s"processando a
mensagem $dado!"))
    testador.isAlive should be(true)
```

```
  }

  "O ator paravel" should "processar mensagens de
finalizacao e encerrar" in {

    val testador: BehaviorTestKit[Mensagem] =
BehaviorTestKit[Mensagem](StoppableActor.behavior())

    assertThrows[ExcecaoDeFinalizacao] {
      testador.run(MensagemFinalizar())
    }

  }

}
```

Nesse código, vemos diversos novos conceitos, tanto do `scalatest` quanto do nosso kit de desenvolvimento de testes. Para criar uma classe com cenários de testes, basta estendermos a classe `AnyFlatSpec` e implementarmos a trait `should.Matchers`.

A classe `AnyFlatSpec` nos fornece um dos chamados *estilos* de implementar testes no `scalatest`, o `FlatSpec`. Nesse estilo, escrevemos os testes em um modelo que pode ser visualizado como se estivéssemos descrevendo os cenários como frases de texto, evidentemente, se utilizarmos o código em inglês, o idioma da biblioteca.

Se observarmos os nomes dos métodos da classe que acabamos de criar, podemos ver que são como frases. Se traduzirmos a palavra *should* presente no código para a palavra *deve*, podemos ver que foram frases completas:

Já a trait `should.Matchers` nos fornece suporte a assertivas. Assertivas são instruções de validação com as quais validamos se os resultados após a execução do teste estão da forma que esperamos. A linha `testador.isAlive should be(true)` é um exemplo de assertiva.

Para utilizar o kit de desenvolvimento de testes, primeiro o instanciamos através do objeto `BehaviorTestKit`. Esse objeto recebe como parâmetro de entrada o ator que será testado. Perceba que, como dito anteriormente, testes síncronos não suportam recursos de um servidor de atores como supervisores, por isso tivemos que fornecer o nosso ator sem o seu supervisor, através do método `behavior`.

Com o kit instanciado, vamos testar o envio de uma mensagem. Para isso, chamamos o método `run` enviando uma mensagem para o ator. Após o envio da mensagem, fazemos assertivas de que a execução ocorreu com sucesso. O método `logEntries` nos permite validar se mensagens de logging foram geradas pelo ator - se esse ator tiver

utilizado o logger fornecido pelo contexto - e o método `isAlive` apenas valida que o nosso ator ainda está ativo como esperado, pois nosso ator só é parado para mensagens do tipo `MensagemFinalizar`.

O outro cenário testa a mensagem de finalização, sendo bastante análogo ao nosso outro cenário de testes. A novidade fica por conta da assertiva `assertThrows`. Se o envio de mensagens para os nossos atores no formato síncrono lança uma exceção, essa exceção é lançada pelo nosso kit. Com essa assertiva, validamos se a exceção foi lançada ou não, terminando o teste com falha caso a esperada exceção não ocorra.

Executando os testes pela nossa IDE ou através do comando `sbt test` no terminal, podemos ver que os testes executaram com sucesso:

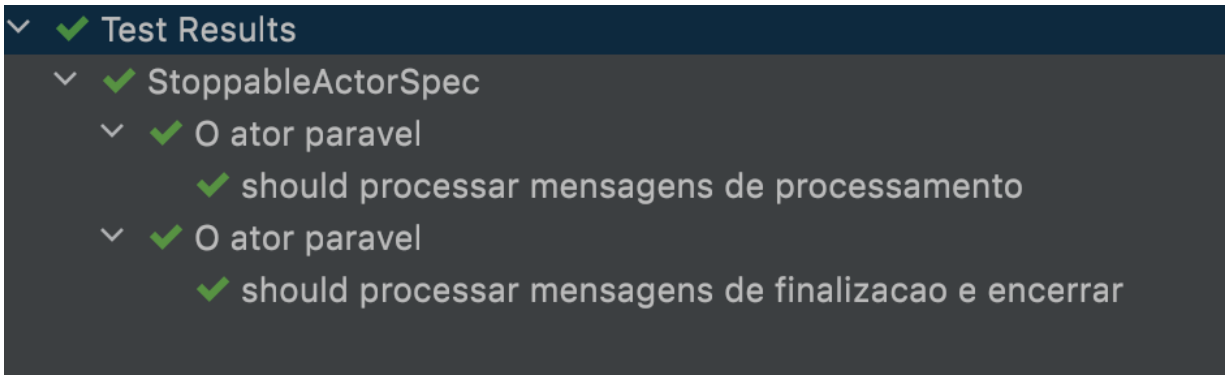



Figura 9.1: Resultado da execução dos testes do ator parável.

Agora, vamos fazer os testes do nosso ator chamador, o ator que servia de intermediador entre o nosso código principal e o ator parável. Para os testes, criamos a classe `CallerActorSpec` com o seguinte conteúdo:

```
package com.casadocodigo.actors.synchronous

import akka.actor.testkit.typed.CapturedLogEvent
import
akka.actor.testkit.typed.scaladsl.BehaviorTestKit
import com.casadocodigo.actors.CallerActor.
{MensagemChamadora, MensagemSolicitarFinalizacao,
MensagemSolicitarProcessamento}
import com.casadocodigo.actors.CallerActor
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should
import org.slf4j.event.Level

class CallerActorSpec extends AnyFlatSpec with
should.Matchers {

  "O ator chamador" should "processar mensagens de
processamento" in {

    val testador: BehaviorTestKit[MensagemChamadora] =
BehaviorTestKit[MensagemChamadora](CallerActor.apply())
    val dado = "Testando!"

    testador.run(MensagemSolicitarProcessamento(dado))
    testador.logEntries() shouldBe
Seq(CapturedLogEvent(Level.INFO, s"solicitando o
processamento da mensagem $dado!"))

    testador.isAlive should be(true)
```

```
  }

  "O ator chamador" should "processar mensagens de
finalizacao e encerrar" in {

    val testador: BehaviorTestKit[MensagemChamadora] =
BehaviorTestKit[MensagemChamadora](CallerActor.apply())

    testador.run(MensagemSolicitarFinalizacao())

    testador.logEntries() shouldBe
Seq(CapturedLogEvent(Level.INFO, s"solicitando a
finalização do processamento!"))
    testador.isAlive should be(true)

  }

}
```

O código dessa classe utiliza os mesmos conceitos da anterior, por isso maiores explicações não são necessárias. Se executarmos os testes, veremos que eles executam corretamente, assim como no caso anterior:

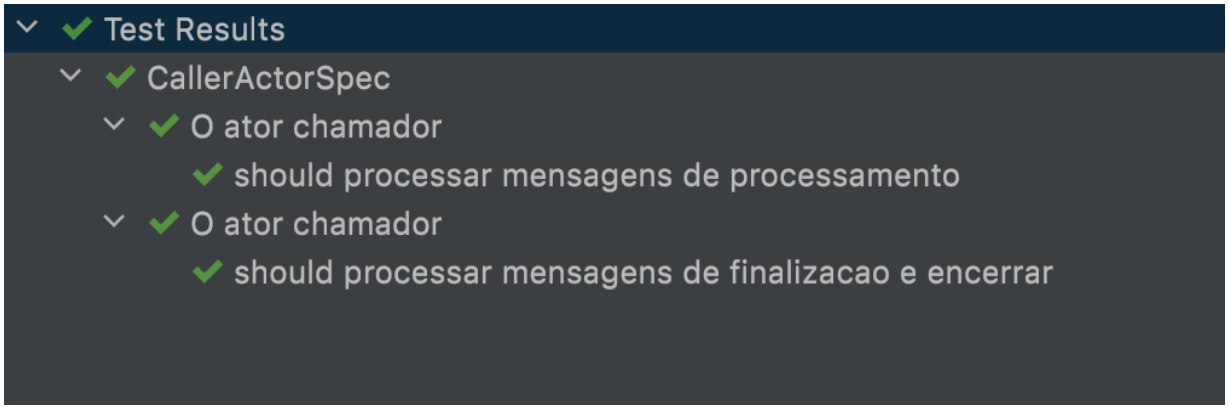



Figura 9.2: Resultado da execução dos testes do ator chamador.

No projeto original do capítulo 4, não tínhamos nenhum ator seguindo o modelo *Ask*, pois apresentamos esse conceito no capítulo seguinte. Vamos abrir uma exceção e criar um novo ator no nosso projeto, apenas para poder demonstrar como realizar testes desse tipo de ator.

Vamos criar o ator `AskActor`. Ele é bastante simples, pois se resume a um simples recebedor de mensagens, gerando um registro de log e devolvendo uma resposta para o ator passado na requisição:

```
package com.casadocodigo.actors

import akka.actor.typed.scaladsl.Behaviors
import akka.actor.typed.{ActorRef, Behavior}

object AskActor {
```

```
  trait MensagemDeRequisicao

  trait MensagemDeResposta

  case class Requisicao(mensagem: String, replyTo:
ActorRef[MensagemDeResposta]) extends
MensagemDeRequisicao

  case class Resposta() extends MensagemDeResposta

  def apply(): Behavior[MensagemDeRequisicao] =
Behaviors.receive[MensagemDeRequisicao] {
    (contexto, mensagem) =>
      mensagem match {
        case Requisicao(mensagem, replyTo) =>
          contexto.log.info(s"processando a mensagem
$mensagem!")
          replyTo ! Resposta()
          Behaviors.same
      }

  }

}
```

Para testar esse ator, vamos criar a classe `AskActorSpec`. Para testar o recebimento da resposta que o nosso ator vai emitir, utilizamos o objeto `TestInbox`. Esse objeto nos fornece um ator observável que podemos fornecer para o nosso ator através do método `ref`.

Após enviarmos uma mensagem para o ator, utilizamos o método `expectMessage`, que consiste de uma assertiva que valida se o ator fornecido pelo

objeto `TestInbox` recebeu uma determinada mensagem.

O código a seguir cria o teste para o nosso novo ator:

```
package com.casadocodigo.actors.synchronous

import akka.actor.testkit.typed.CapturedLogEvent
import akka.actor.testkit.typed.scaladsl.
{BehaviorTestKit, TestInbox}
import com.casadocodigo.actors.AskActor.
{MensagemDeRequisicao, MensagemDeResposta, Requisicao,
Resposta}
import com.casadocodigo.actors.AskActor
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should
import org.slf4j.event.Level

class AskActorSpec extends AnyFlatSpec with
should.Matchers {

  "O ator responsivo" should "processar mensagens e nos
retornar uma resposta" in {

    val testador: BehaviorTestKit[MensagemDeRequisicao]
= BehaviorTestKit[MensagemDeRequisicao]
(AskActor.apply())
    val dado = "Testando!"
    val recebedor = TestInbox[MensagemDeResposta]()

    testador.run(Requisicao(dado, recebedor.ref))

    testador.logEntries() shouldBe
Seq(CapturedLogEvent(Level.INFO, s"processando a
mensagem $dado!"))
```

```
    recebedor.expectMessage(Resposta())
    testador.isAlive should be(true)

  }

}
```

E após executarmos os testes, podemos ver novamente que a execução valida com sucesso que o nosso ator funciona como o esperado:

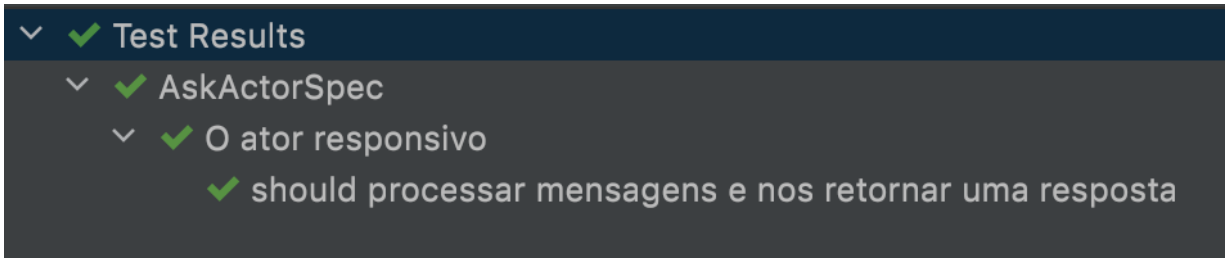



Figura 9.3: Resultado da execução dos testes do ator responsivo (Ask).

Agora que temos coberta a base para a criação de testes síncronos, vamos explorar o nosso outro formato, testando os nossos atores no formato assíncrono.

**Testes assíncronos**

Agora, vamos explorar a construção de testes no formato assíncrono. Nesse formato, conforme dito anteriormente, temos um ambiente mais próximo do

ambiente real de execução da nossa aplicação, com todos os recursos disponíveis para serem testados.

Vamos começar criando a classe `StoppableActorSpec` - é o mesmo nome da nossa classe de testes síncronos, porém neste caso dentro do pacote `com.casadocodigo.actors.asynchronous` -, com o seguinte conteúdo:

```
package com.casadocodigo.actors.asynchronous

import akka.actor.testkit.typed.scaladsl.{ActorTestKit,
LoggingTestKit}
import com.casadocodigo.actors.StoppableActor
import com.casadocodigo.actors.StoppableActor.
{MensagemFinalizar, MensagemProcessamento}
import org.scalatest._
import org.scalatest.flatspec._
import org.scalatest.matchers._


class StoppableActorSpec extends AnyFlatSpec with
BeforeAndAfterAll with should.Matchers {

  val testador = ActorTestKit()
  implicit val system = testador.system
  val ator = testador.spawn(StoppableActor(),
"StoppableActor")

  override def afterAll(): Unit =
testador.shutdownTestKit()

  "O ator paravel" should "processar mensagens de
processamento" in {
```

```
    val dado = "Testando!"

    LoggingTestKit.info(s"processando a mensagem
$dado!").expect {
      ator ! MensagemProcessamento(dado)
    }

  }

  "O ator paravel" should "processar mensagens de
finalizacao" in {

    LoggingTestKit.info(s"finalizando o
processamento!").expect {
      ator ! MensagemFinalizar()
    }
    val dado = "Testando!"
    LoggingTestKit.info(s"processando a mensagem
$dado!").expect {
      ator ! MensagemProcessamento(dado)
    }

  }

  "O ator paravel" should "processar mensagens de
finalizacao e encerrar" in {

    testador.stop(ator)
    val outroAtor =
testador.spawn(StoppableActor.behavior(),
"OtherStoppableActor")
    val recebedorDeadLetter =
testador.createDeadLetterProbe()

    LoggingTestKit.info(s"finalizando o
processamento!").expect {
```

```
      outroAtor ! MensagemFinalizar()
    }

    recebedorDeadLetter.expectTerminated(outroAtor)
  }

}
```

Como podemos ver, não é muito diferente realizar testes síncronos de assíncronos. Criamos o servidor de testes de atores através do objeto `ActorTestKit`, que permite instanciarmos atores de forma bastante parecida com que fazemos com um servidor de atores tradicional. Quando os nossos testes acabarem, precisamos encerrar o servidor, por isso importamos a trait `BeforeAndAfterAll`.

Essa trait fornece os métodos `beforeAll` e `afterAll`, que, respectivamente, são executados antes e depois de todos os testes da classe de testes. No nosso exemplo, utilizamos o método `afterAll`, com o qual encerramos o servidor depois que todos os testes tiverem acabado através do método `shutdownTestKit`.

Outra mudança nesse formato é em como verificamos se mensagens de log foram geradas, através do objeto `LoggingTestKit`. Esse objeto utiliza técnicas de programação orientada a aspectos, que fogem do escopo do nosso estudo, mas basta compreender que todo logging gerado

pelo código dentro da função passada por parâmetro ao objeto - o código entre chaves ( `{...}` ) - será analisado e uma assertiva será feita para verificar se alguma das mensagens de log é igual a que passamos também por parâmetro. Caso não encontre nenhum registro da mensagem passada, uma exceção será lançada.

O nosso terceiro teste, o teste `"O ator paravel" should "processar mensagens de finalizacao e encerrar"` também merece uma explicação especial. Nesse método, paramos o ator que criamos através do método `spawn` no início do teste, e criamos o ator novamente, porém utilizando o método `behavior()` , tal qual fizemos no teste síncrono. Ora, não dissemos que a vantagem dos testes assíncronos é que podemos testar configurações de recursos como supervisores?

É exatamente por estarmos testando o supervisor que estamos fazendo isso. No teste anterior da classe, o teste `"O ator paravel" should "processar mensagens de finalizacao"` , fizemos um teste onde enviamos uma mensagem de processamento logo após a mensagem de finalização. Isso testa que o nosso supervisor funcionou corretamente, pois logo após solicitar uma finalização, o ator foi reiniciado e já estava disponível para processar mensagens novamente.

No caso do terceiro teste, instanciamos um novo ator sem o uso das configurações de supervisão, apenas para garantir que ele está de fato sendo parado. Assim, após instanciarmos o ator, criamos um objeto escutador - chamado na terminologia do kit de testes de `probe` - através do método `createDeadLetterProbe`. Esse escutador observará durante alguns segundos, após enviarmos a mensagem de finalização, se o nosso ator será encerrado, através do método `expectTerminated`.

Ao executarmos os testes, podemos observar que todos os testes executaram com sucesso conforme o esperado:

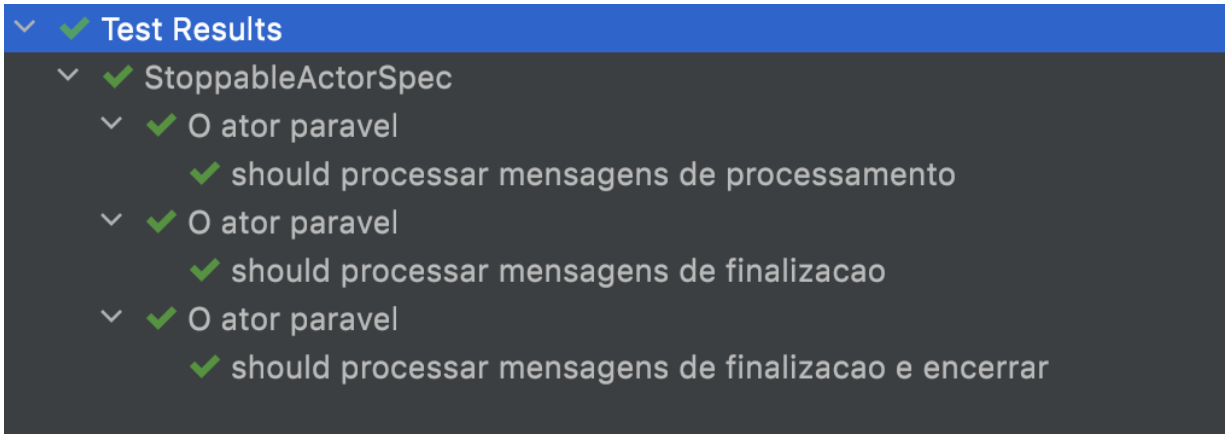



Figura 9.4: Resultado da execução dos testes assíncronos do ator parável.

A seguir, vamos criar os testes do nosso ator chamador. Eles utilizam os mesmos recursos dos testes anteriores, sem novidades:

```
package com.casadocodigo.actors.asynchronous

import akka.actor.testkit.typed.scaladsl.{ActorTestKit,
LoggingTestKit}
import com.casadocodigo.actors.CallerActor
import com.casadocodigo.actors.CallerActor.
{MensagemSolicitarFinalizacao,
MensagemSolicitarProcessamento}
import org.scalatest.BeforeAndAfterAll
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should

class CallerActorSpec extends AnyFlatSpec with
BeforeAndAfterAll with should.Matchers {

  val testador = ActorTestKit()
  implicit val system = testador.system
  val ator = testador.spawn(CallerActor(),
"CallerActor")

  override def afterAll(): Unit =
testador.shutdownTestKit()

  "O ator chamador" should "processar mensagens de
processamento" in {

    val dado = "Testando!"

    LoggingTestKit.info(s"solicitando o processamento
da mensagem $dado!").expect {
      ator ! MensagemSolicitarProcessamento(dado)
    }

  }

  "O ator chamador" should "processar mensagens de
finalizacao e encerrar" in {
```

```
    LoggingTestKit.info(s"solicitando a finalização do
processamento!").expect {
      ator ! MensagemSolicitarFinalizacao()
    }

  }

}
```

Ao executarmos os testes, podemos observar que todos os testes executaram com sucesso conforme o esperado:

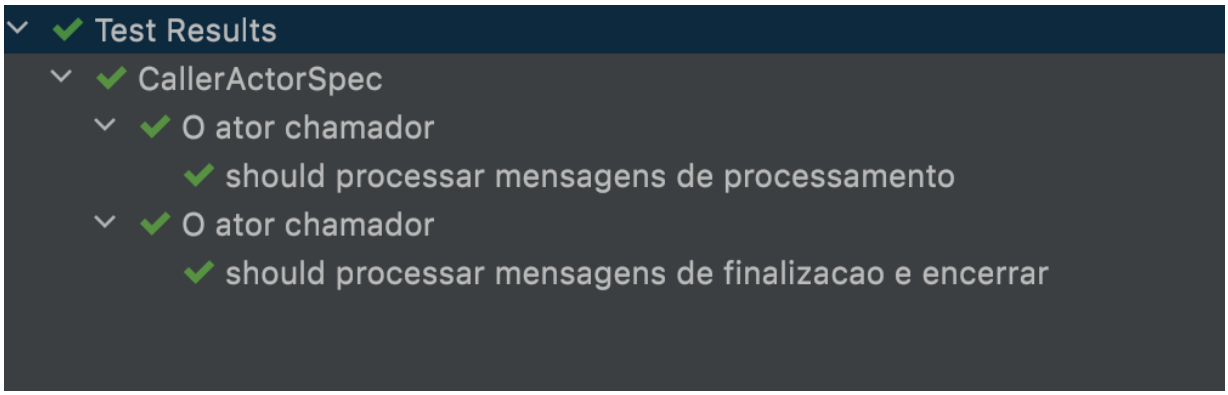



Figura 9.5: Resultado da execução dos testes assíncronos do ator chamador.

Por fim, vamos fazer a versão assíncrona do teste do ator modelo *Ask*, que criamos anteriormente. Para testarmos o recebimento de uma resposta, desta vez criamos um recebedor utilizando o método `createTestProbe` e ficamos esperando pela resposta através do método `expectMessage` :

```
package com.casadocodigo.actors.asynchronous

import akka.actor.testkit.typed.scaladsl.{ActorTestKit,
LoggingTestKit}
import com.casadocodigo.actors.AskActor
import com.casadocodigo.actors.AskActor.
{MensagemDeResposta, Requisicao, Resposta}
import org.scalatest.BeforeAndAfterAll
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should

class AskActorSpec extends AnyFlatSpec with
BeforeAndAfterAll with should.Matchers {

  val testador = ActorTestKit()
  implicit val system = testador.system
  val ator = testador.spawn(AskActor(), "AskActor")

  override def afterAll(): Unit =
testador.shutdownTestKit()

  "O ator responsivo" should "processar mensagens e nos
retornar uma resposta" in {

    val dado = "Testando!"
    val recebedor =
testador.createTestProbe[MensagemDeResposta]()

    LoggingTestKit.info(s"processando a mensagem
$dado!").expect {
      ator ! Requisicao(dado, recebedor.ref)
    }

    recebedor.expectMessage(Resposta())
  }

}
```

E após executarmos os testes, podemos ver novamente que a execução valida com sucesso que o nosso ator funciona como o esperado:

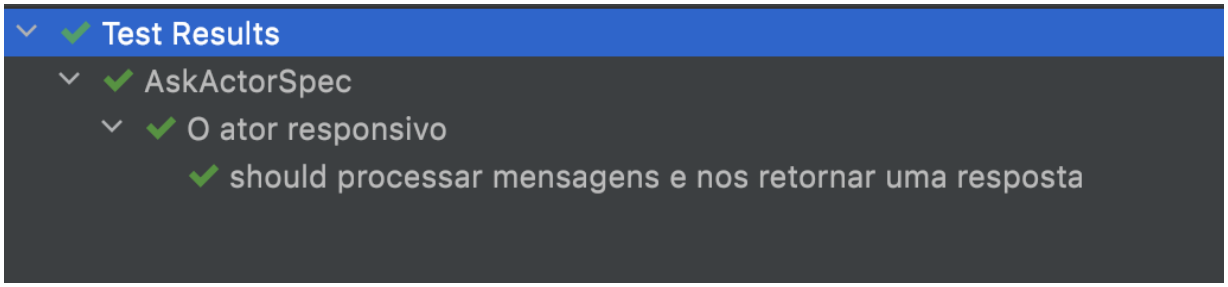



Figura 9.6: Resultado da execução dos testes assíncronos do ator responsivo (Ask).

E assim concluímos nossa exploração pelas funcionalidades básicas principais do kit de desenvolvimento de testes do Akka. A seguir, vamos dar uma olhada no kit para desenvolvimento de testes do Akka Streams.

## 9.2 Testes no Akka Streams

Agora que aprendemos como realizar testes no Akka, vamos criar testes no Akka Streams. Assim sendo, vamos copiar o nosso projeto iniciado no capítulo 7 e modificar o arquivo `build.sbt` incluindo as dependências necessárias para criarmos os testes:

```
name := "akka-streams"

version := "1.0"

scalaVersion := "2.13.6"
val AkkaVersion = "2.6.15"
val slickVersion = "3.3.3"
val akkaHttpVersion = "10.2.6"

mainClass in Compile := Some("com.casadocodigo.Boot")

//Akka e outras dependencias
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor-typed" %
AkkaVersion,
  "com.typesafe.akka" %% "akka-stream" % AkkaVersion,
  "com.typesafe.akka" %% "akka-http" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-spray-json" %
akkaHttpVersion,
  "com.typesafe.akka" %% "akka-actor-testkit-typed" %
AkkaVersion % Test,
  "com.typesafe.akka" %% "akka-stream-testkit" %
AkkaVersion % Test,
  "org.scalatest" %% "scalatest" % "3.2.9" % Test,
  "org.scalamock" %% "scalamock" % "5.1.0" % Test
)

// dependencias de logging
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-slf4j" % AkkaVersion,
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)

// dependencias do Alpakka
libraryDependencies ++= Seq(
  "com.lightbend.akka" %% "akka-stream-alpakka-file" %
```

```
"3.0.3",
  "com.typesafe.akka" %% "akka-stream-kafka" % "2.1.1",
  "com.lightbend.akka" %% "akka-stream-alpakka-slick" %
"3.0.3",
  "com.lightbend.akka" %% "akka-stream-alpakka-csv" %
"3.0.3"
)

// dependencias de banco
libraryDependencies ++= Seq(
  "org.postgresql" % "postgresql" % "42.2.23"
)
```

Além do já utilizado `scalatest`, estamos incluindo no projeto as bibliotecas `akka-stream-testkit` e `scalamock`. O `akka-stream-testkit`, como o próprio nome sugere, é o kit de desenvolvimento de testes do Akka Streams. Já o `scalamock` nos permite criar *mocks*.

Mocks são como "emuladores de código" que podemos utilizar para simular partes do código que não podemos utilizar no ambiente de testes. No nosso exemplo, utilizaremos mocks para simular as chamadas de APIs de uma de nossas streams.

**Testando nossa primeira stream**

Vamos começar construindo o teste da stream `PrimeiraStream`. Para isso, vamos criar a classe `PrimeiraStreamSpec` com o seguinte conteúdo:

```
package com.casadocodigo.streams

import akka.actor.testkit.typed.scaladsl.ActorTestKit
import akka.actor.typed.ActorSystem
import akka.stream.testkit.scaladsl.TestSink
import org.scalatest.BeforeAndAfterAll
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should


class PrimeiraStreamSpec extends AnyFlatSpec with
BeforeAndAfterAll with should.Matchers {

  private val testador = ActorTestKit()
  private implicit val system: ActorSystem[Nothing] =
testador.system

  override def afterAll(): Unit =
testador.shutdownTestKit()

  "A primeira stream" should "transformar uma sequencia
de numeros em uma sequencia de strings" in {
    PrimeiraStream
      .done
      .runWith(TestSink[String]())
      .request(2)
      .expectNext("sou o numero 2", "sou o numero 4")
  }

}
```

Nesse teste, vemos o primeiro componente que utilizamos do kit de desenvolvimento de testes, o objeto `TestSink` . Através desse objeto, conseguimos monitorar nossas streams através de

suas fozes, observando as mensagens que chegam do processamento. No nosso exemplo, fazemos com que o objeto `TestSink` solicite o envio de duas mensagens - a partir do método `request` - e criamos uma assertiva com a qual verificamos como são as duas mensagens que solicitamos, através do método `expectNext` .

Se relembrarmos o nosso código original da stream, temos um problema: todo o código da stream está dentro do método `primeiraStream` . Dessa forma, não é possível extrair a stream para realizarmos os testes, como podemos ver a seguir relembrando o código original:

```
package com.casadocodigo.streams

import akka.{Done, NotUsed}
import akka.stream.scaladsl.Source
import com.casadocodigo.Boot.{ec, system}

import scala.concurrent.Future

object PrimeiraStream {

  def primeiraStream(): Unit = {
    val source: Source[Int, NotUsed] = Source(1 to 100)
    val done: Future[Done] = source
      .filter(i => i % 2 == 0)
      .map(i => f"sou o numero $i")
      .runForeach(i => println(i))

    done.onComplete(_ => println("terminando a
```

```
execução!"))
  }

}
```

Isso nos traz uma importante lição: TDD é bom! Vamos observar nesta seção que vamos refatorar para expor streams, ou pedaços de streams, de modo que seja possível efetuar os testes e que a estrutura das streams fique melhor componentizada ao quebrá-la em diferentes métodos. Tais refatorações não eram visíveis quando estávamos desenvolvendo a primeira versão de nossas streams, pois a nossa visão era a visão "de dentro" do ponto de vista interno do sistema. Testes são como se fossem os primeiros clientes da nossa aplicação. Eles nos dão uma visão de interface do que está sendo exposto pelo sistema, permitindo-nos enxergar esses tipos de melhorias.

Assim, refatoramos o código dividindo a stream em variáveis e expondo a variável `done` para a classe de testes. Podemos realizar o teste:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.stream.scaladsl.Source
import com.casadocodigo.Boot.{ec, system}
```

```
object PrimeiraStream {

  private val source: Source[Int, NotUsed] = Source(1
to 100)
  private[streams] val done: Source[String, NotUsed] =
source
    .filter(i => i % 2 == 0)
    .map(i => f"sou o numero $i")

  def primeiraStream(): Unit = {
    done.runForeach(i => println(i))
        .onComplete(_ => println("terminando a
execução!"))
  }

}
```

Reparou no `private[streams]` ? Em Scala, temos um modificador especial de acessibilidade de métodos, classes etc., com o qual podemos expor atributos privados ampliando o acesso a eles também no nível de pacote. No nosso caso, a variável `done` é privada com acesso de pacote ao pacote `streams` . Isso significa que apenas o objeto `PrimeiraStream` e classes e objetos dentro do pacote `streams` podem acessar a variável. Como a classe de testes `PrimeiraStreamSpec` também se encontra no pacote `streams` , isso significa que ela também tem acesso à variável `done` .

Ao executarmos os testes, podemos ver que a execução é um sucesso, validando a nossa stream:

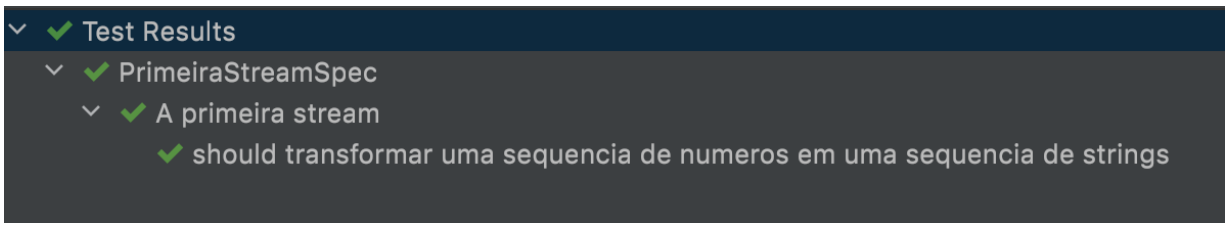

Test Results
PrimeiraStreamSpec
A primeira stream
should transformar uma sequencia de numeros em uma sequencia de strings

Figura 9.7: Resultado da execução dos testes da primeira stream.

## Testando nossa stream do sisteam de arquivos para o Apache Kafka

A seguir, vamos criar o teste da nossa stream que lê de arquivos para persistir no Kafka. Para esse teste, primeiro refatoramos o código da aplicação, de modo que deixemos exposta a parte que efetua a transformação dos dados vindos do arquivo que serão gravados no Kafka.

Você pode notar um padrão na forma em que estamos desenvolvendo os testes: utilizamos mocks e excluímos componentes externos, como o Kafka e o sistema de arquivos dos testes. Isso acontece porque os kits de desenvolvimento de testes têm por objetivo auxiliar na criação de *testes unitários*, ou seja, testes que validam lógica e regras de aplicação, e não a configuração de componentes de terceiros que apenas configuramos para utilização. Inclusive, existem famosas discussões sobre se faz sentido mesmo realizar testes desses componentes na sua aplicação, dado que eles mesmos já possuem testes em seus respectivos projetos de origem. Testes que envolvam a subida de bancos de dados, APIs etc. para a execução dos chamados *testes de integração* não se utilizam de nenhum recurso especial fornecido pelo framework, simplesmente consistindo em subir a aplicação e validar a sua execução. Por essa razão, esse tipo de teste não faz parte do escopo deste livro.

Para efeito de comparação, este é o código original da stream:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.kafka.ProducerSettings
```

```
import akka.kafka.scaladsl.Producer
import akka.stream.alpakka.file.DirectoryChange
import akka.stream.alpakka.file.scaladsl.{Directory,
DirectoryChangesSource, FileTailSource}
import akka.stream.scaladsl.Source
import com.casadocodigo.Boot.{system, config}
import org.apache.kafka.clients.producer.ProducerRecord
import
org.apache.kafka.common.serialization.StringSerializer

import java.io.FileNotFoundException
import java.nio.file.{FileSystems, Path}
import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

object ArquivoParaKafka {

  private val kafkaProducerSettings =
    ProducerSettings.create(system, new
StringSerializer(), new StringSerializer())

.withBootstrapServers(config.getString("bootstrapServer
s"))

  private val sistemaDeArquivos =
FileSystems.getDefault
  private val diretorio = "./input_dir"
  private val mudancasNoDiretorio =
DirectoryChangesSource(sistemaDeArquivos.getPath(direto
rio), pollInterval = 1 second, maxBufferSize = 1000)
  private val diretorioInicial: Source[Path, NotUsed] =
Directory.ls(sistemaDeArquivos.getPath(diretorio))

  def iniciarStreams(): Unit = {
    diretorioInicial.runForeach {
      path =>
```

```
obterLeitorDeArquivo(path).merge(obterVerificadorDeArqu
ivoDeletado(path), eagerComplete = true)
          .map(value => new ProducerRecord[String,
String]("contas", value))

.runWith(Producer.plainSink(kafkaProducerSettings))
    }
    mudancasNoDiretorio.runForeach {
      case (path, change) =>
        change match {
          case DirectoryChange.Creation =>

obterLeitorDeArquivo(path).merge(obterVerificadorDeArqu
ivoDeletado(path), eagerComplete = true)
              .map(value => new ProducerRecord[String,
String]("contas", value))

.runWith(Producer.plainSink(kafkaProducerSettings))
        }
    }
  }


  private def obterVerificadorDeArquivoDeletado(path:
Path): Source[Nothing, NotUsed] =
DirectoryChangesSource(path.getParent, 1 second, 8192)
    .collect {
      case (p, DirectoryChange.Deletion) if path == p
=>
        throw new FileNotFoundException(path.toString)
    }
    .recoverWithRetries(1, {
      case _: FileNotFoundException => Source.empty
    })

  private def obterLeitorDeArquivo(path: Path):
Source[String, NotUsed] = FileTailSource.lines(
```

```
    path = path,
    maxLineSize = 8192,
    pollingInterval = 250.millis
  )

}
```

Este é o código refatorado:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.kafka.ProducerSettings
import akka.kafka.scaladsl.Producer
import akka.stream.alpakka.file.DirectoryChange
import akka.stream.alpakka.file.scaladsl.{Directory,
DirectoryChangesSource, FileTailSource}
import akka.stream.scaladsl.{Flow, Source}
import com.casadocodigo.Boot.{config, system}
import com.casadocodigo.streams.KafkaParaBanco.Conta
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.clients.producer.ProducerRecord
import
org.apache.kafka.common.serialization.StringSerializer

import java.io.FileNotFoundException
import java.nio.file.{FileSystems, Path}
import scala.concurrent.duration.DurationInt
import scala.language.postfixOps

object ArquivoParaKafka {

  private val sistemaDeArquivos =
FileSystems.getDefault
  private val diretorio = "./input_dir"
  private val mudancasNoDiretorio =
DirectoryChangesSource(sistemaDeArquivos.getPath(direto
```

```
rio), pollInterval = 1 second, maxBufferSize = 1000)
    .filter {
      case (_, change) =>
        change == DirectoryChange.Creation
    }
    .map {
      case (path, _) =>

obterLeitorDeArquivo(path).merge(obterVerificadorDeArqu
ivoDeletado(path), eagerComplete = true)
          .via(fluxoDeTransformacaoDados())
    }
  private[streams] val diretorioInicial:
Source[Source[ProducerRecord[String, String], NotUsed],
NotUsed] =
Directory.ls(sistemaDeArquivos.getPath(diretorio))
    .map(path =>

obterLeitorDeArquivo(path).merge(obterVerificadorDeArqu
ivoDeletado(path), eagerComplete = true)
        .via(fluxoDeTransformacaoDados())
    )

  def fluxoDeTransformacaoDados(): Flow[String,
ProducerRecord[String, String], NotUsed] = {
    Flow[String].map(value => new
ProducerRecord[String, String]("contas", value))
  }


  def iniciarStreams(): Unit = {
    val kafkaProducerSettings =
      ProducerSettings.create(system, new
StringSerializer(), new StringSerializer())

.withBootstrapServers(config.getString("bootstrapServer
s"))
```

```
    diretorioInicial
      .runForeach(record =>

record.runWith(Producer.plainSink(kafkaProducerSettings
)
        )
      )

    mudancasNoDiretorio
      .runForeach(record =>

record.runWith(Producer.plainSink(kafkaProducerSettings
)
        )
      )
  }


  private def obterVerificadorDeArquivoDeletado(path:
Path): Source[Nothing, NotUsed] =
DirectoryChangesSource(path.getParent, 1 second, 8192)
    .collect {
      case (p, DirectoryChange.Deletion) if path == p
=>
        throw new FileNotFoundException(path.toString)
    }
    .recoverWithRetries(1, {
      case _: FileNotFoundException => Source.empty
    })

  private def obterLeitorDeArquivo(path: Path):
Source[String, NotUsed] = FileTailSource.lines(
    path = path,
    maxLineSize = 8192,
    pollingInterval = 250.millis
  )
```

```
}
```

E este é o código da classe de teste, chamada `ArquivoParaKafkaSpec`. Nela, temos o nosso primeiro contato com outro componente do kit de desenvolvimento de testes do Akka Streams, o objeto `TestSource`. Com ele, conseguimos testar fluxos de streams encaixando-os como o "miolo" de uma stream composta por um `TestSource` e um `TestSink`, como fizemos no trecho:

```
val (entrada, saida) = TestSource.probe[String]
      .via(fluxoDeTransformacaoDados())
      .toMat(TestSink[ProducerRecord[String, String]]
())(Keep.both).run()
```

Esse trecho nos retorna a nascente e a foz da stream dentro das variáveis `entrada` e `saida`, respectivamente. A seguir, como veremos na classe de testes completa, basta enviarmos mensagens - utilizando o método `sendNext` - e verificamos o recebimento dos resultados na saída. Para a comparação dos resultados, utilizamos o método `expectNextUnordered`, indicando que estamos interessados nas mensagens sendo recebidas, não importando a ordem em que elas são processadas.

O código a seguir compõe toda a classe de teste:

```
package com.casadocodigo.streams

import akka.actor.ActorSystem
import akka.stream.scaladsl.Keep
import akka.stream.testkit.scaladsl.{TestSink,
TestSource}
import org.apache.kafka.clients.producer.ProducerRecord
import org.scalatest.BeforeAndAfterAll
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should
import
com.casadocodigo.streams.ArquivoParaKafka.fluxoDeTransf
ormacaoDados

import scala.language.postfixOps

class ArquivoParaKafkaSpec extends AnyFlatSpec with
BeforeAndAfterAll with should.Matchers {

  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")

  override def afterAll(): Unit = system.terminate()

  "A stream de arquivos para o kafka" should "enviar
linhas do arquivo para persistencia no kafka" in {

    val (entrada, saida) = TestSource.probe[String]
      .via(fluxoDeTransformacaoDados())
      .toMat(TestSink[ProducerRecord[String, String]]
())(Keep.both).run()
    saida.request(2)
    entrada.sendNext("\"Alexandre\",39,112322211")
    entrada.sendNext("\"Ana
Carolina\",\"18\",98274391")
    saida.expectNextUnordered(new
ProducerRecord("contas", "\"Alexandre\",39,112322211"),
```

```
      new ProducerRecord("contas", "\"Ana
Carolina\",\"18\",98274391"))
  }

}
```

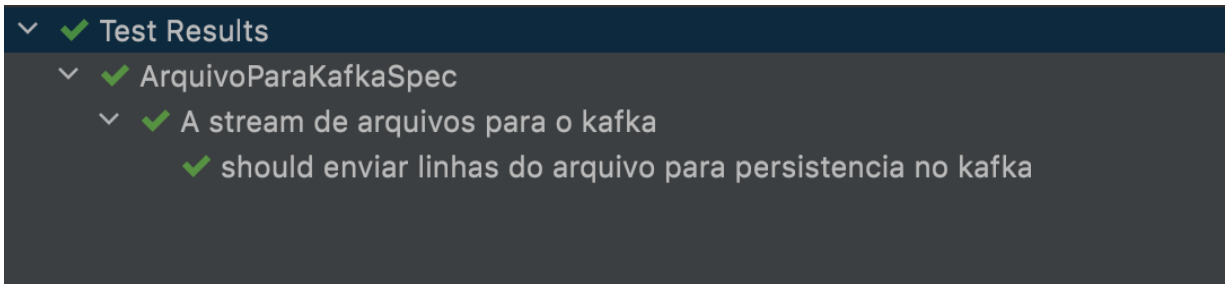

Test Results
ArquivoParaKafkaSpec
A stream de arquivos para o kafka
should enviar linhas do arquivo para persistencia no kafka

Figura 9.8: Resultado da execução dos testes da stream de arquivo para Kafka.

## Testando nossa stream do Apache Kafka para a base de dados relacional

Continuando, agora criaremos o teste da stream que lê do Kafka e persiste no banco de dados. Analogamente ao que fizemos anteriormente, vamos refatorar a stream para expor o "miolo" da stream, no qual está a transformação dos dados vindos do Kafka antes de irem para o banco. Este é o código original da classe:

```
package com.casadocodigo.streams

import akka.kafka.{ConsumerSettings, Subscriptions}
import akka.kafka.scaladsl.Consumer
import akka.stream.alpakka.csv.scaladsl.{CsvParsing,
```

```
CsvToMap}
import akka.stream.alpakka.slick.scaladsl.Slick
import akka.util.ByteString
import org.apache.kafka.clients.consumer.ConsumerConfig
import
org.apache.kafka.common.serialization.StringDeserialize
r
import com.casadocodigo.Boot.{config, system, session}
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import java.nio.charset.StandardCharsets

object KafkaParaBanco {

  case class Conta(nome: String, idade: Int, documento:
Long)

  class ContaSchema(tag: Tag) extends Table[Conta](tag,
"conta") {
    def nome = column[String]("nome")

    def idade = column[Int]("idade")

    def documento = column[Long]("documento")

    def * = (nome, idade, documento) <> (Conta.tupled,
Conta.unapply)
  }

  private val db = session.db
  private val tabela = TableQuery[ContaSchema]
  db.run(DBIO.seq(
    tabela.schema.createIfNotExists
  ))
```

```
  private val consumerSettings =
    ConsumerSettings(system, new StringDeserializer,
new StringDeserializer)

.withBootstrapServers(config.getString("bootstrapServer
s"))
      .withGroupId("grupo")

.withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest")

.withProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
"true")

.withProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CO
NFIG, "5000")

  def iniciarStreams(): Unit = {
    Consumer
      .plainSource(
        consumerSettings,
        Subscriptions.topics("contas")
      ).map(registro => {
      val linhaComQuebraDelinha = registro.value() +
"\n"
      ByteString(linhaComQuebraDelinha)
    }
    )
      .via(CsvParsing.lineScanner())

.via(CsvToMap.withHeadersAsStrings(StandardCharsets.UTF
_8, "nome", "idade", "documento"))
      .map(registro => Conta(registro("nome"),
registro("idade").toInt, registro("documento").toLong))
      .runWith(
        Slick.sink(conta => sqlu"INSERT INTO conta
VALUES(${conta.nome}, ${conta.idade},
```

```
${conta.documento})")
      )
  }

}
```

Este é o código refatorado:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.kafka.{ConsumerSettings, Subscriptions}
import akka.kafka.scaladsl.Consumer
import akka.stream.alpakka.csv.scaladsl.{CsvParsing,
CsvToMap}
import akka.stream.alpakka.slick.scaladsl.Slick
import akka.stream.scaladsl.{Flow, Source}
import akka.util.ByteString
import org.apache.kafka.clients.consumer.
{ConsumerConfig, ConsumerRecord}
import
org.apache.kafka.common.serialization.StringDeserialize
r
import com.casadocodigo.Boot.{config, session, system}
import slick.dbio.DBIO
import slick.lifted.Tag
import slick.jdbc.PostgresProfile.api._

import java.nio.charset.StandardCharsets

object KafkaParaBanco {

  case class Conta(nome: String, idade: Int, documento:
Long)

  class ContaSchema(tag: Tag) extends Table[Conta](tag,
"conta") {
```

```
    def nome = column[String]("nome")

    def idade = column[Int]("idade")

    def documento = column[Long]("documento")

    def * = (nome, idade, documento) <> (Conta.tupled,
Conta.unapply)
  }

  def fluxoDeTransformacaoDados():
Flow[ConsumerRecord[String, String], Conta, NotUsed] =
{
    Flow[ConsumerRecord[String, String]].map(registro
=> {
      val linhaComQuebraDelinha = registro.value() +
"\n"
      ByteString(linhaComQuebraDelinha)
    }
    )
      .via(CsvParsing.lineScanner())

.via(CsvToMap.withHeadersAsStrings(StandardCharsets.UTF
_8, "nome", "idade", "documento"))
      .map(registro => Conta(registro("nome"),
registro("idade").toInt, registro("documento").toLong))
  }

  def iniciarStreams(): Unit = {
    val db = session.db
    val tabela = TableQuery[ContaSchema]
    db.run(DBIO.seq(
      tabela.schema.createIfNotExists
    ))

    val consumerSettings =
      ConsumerSettings(system, new StringDeserializer,
```

```
new StringDeserializer)

.withBootstrapServers(config.getString("bootstrapServer
s"))
        .withGroupId("grupo")

.withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest")

.withProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
"true")

.withProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CO
NFIG, "5000")

    Consumer
      .plainSource(
        consumerSettings,
        Subscriptions.topics("contas")
      )
      .via(fluxoDeTransformacaoDados())
      .runWith(
        Slick.sink(conta => sqlu"INSERT INTO conta
VALUES(${conta.nome}, ${conta.idade},
${conta.documento})")
      )
  }

}
```

É possível notar que fizemos outra alteração além de expor o "miolo" da stream: trouxemos algumas variáveis como `db` e `tabela` para dentro do método `iniciarStreams`. A razão para isso é que essas classes demandam que o ambiente produtivo esteja operante, no caso delas, que o banco de dados da aplicação já esteja funcionando. Como não temos um banco de dados no ambiente de teste, movemos essas variáveis para dentro do método, de modo que elas só sejam utilizadas dentro do contexto de execução da aplicação, não dos testes unitários.

Por fim, temos a classe de testes chamada `KafkaParaBancoSpec`. Essa classe é bastante semelhante à já criada para o teste da stream de arquivo para o Kafka, portanto dispensa maiores explicações:

```
package com.casadocodigo.streams

import akka.actor.ActorSystem
import akka.stream.scaladsl.Keep
import akka.stream.testkit.scaladsl.{TestSink,
TestSource}
import com.casadocodigo.streams.KafkaParaBanco.{Conta,
fluxoDeTransformacaoDados}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.scalatest.BeforeAndAfterAll
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should
```

```
import scala.language.postfixOps

class KafkaParaBancoSpec extends AnyFlatSpec with
BeforeAndAfterAll with should.Matchers {

  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")

  override def afterAll(): Unit = system.terminate()

  "A stream do kafka para o banco" should "ler dados do
kafka para persistencia no banco" in {

    val (entrada, saida) =
TestSource.probe[ConsumerRecord[String, String]]
      .via(fluxoDeTransformacaoDados())
      .toMat(TestSink[Conta]())(Keep.both).run()

    saida.request(2)
    entrada.sendNext(new ConsumerRecord[String, String]
("contas", 0,
      0, "chave", "\"Alexandre\",39,112322211"))
    entrada.sendNext(new ConsumerRecord[String, String]
("contas", 0,
      0, "chave", "\"Ana Carolina\",\"18\",98274391"))
    saida.expectNextUnordered(Conta("Alexandre", 39,
112322211),
      Conta("Ana Carolina", 18, 98274391))

  }

}
```

Executando os testes, vemos que conseguimos validar as operações de transformação feitas pela

stream com sucesso:

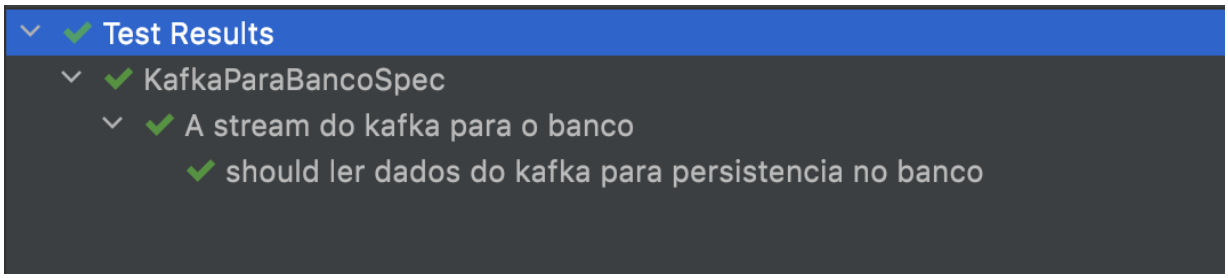



Figura 9.9: Resultado da execução dos testes da stream de Kafka para banco.

## Testando nossa stream do Apache Kafka para APIs REST

Finalmente, vamos criar o teste da nossa última stream, a stream que envia dados do Kafka para as APIs. A refatoração da stream é bastante análoga ao que já fizemos com as outras streams. As principais diferenças são que extraímos as chamadas HTTP das APIs para um método utilitário `chamarHttp` a fim de permitir que criemos mocks desse método. Também declaramos o componente de configurações `config` como implícito nos métodos que montam a stream. Isso permite que instanciemos o nosso próprio componente dentro do nosso teste e utilizemos o componente criado dentro do objeto `Boot` quando estamos executando a aplicação. Mais uma vez para efeito de comparação, este é o código original da stream:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.http.scaladsl.Http
import akka.http.scaladsl.model.{ContentTypes,
HttpEntity, HttpMethods, HttpRequest}
import akka.kafka.scaladsl.Consumer
import akka.kafka.{ConsumerSettings, Subscriptions}
import akka.stream.{ActorAttributes, FlowShape,
Supervision}
import akka.stream.alpakka.csv.scaladsl.{CsvParsing,
CsvToMap}
import akka.stream.scaladsl.{Broadcast, Flow, GraphDSL,
Sink, Source, Zip}
import akka.util.ByteString
import com.casadocodigo.Boot.{config, system}
import com.casadocodigo.streams.KafkaParaBanco.Conta
import org.apache.kafka.clients.consumer.ConsumerConfig
import
org.apache.kafka.common.serialization.StringDeserialize
r
import spray.json._

import java.nio.charset.StandardCharsets
import scala.language.postfixOps
import scala.util.Success

object KafkaParaAPIs extends SerializadorJSON {

  private val decider: Supervision.Decider = _ =>
Supervision.Restart

  private val consumerSettings =
    ConsumerSettings(system, new StringDeserializer,
new StringDeserializer)

.withBootstrapServers(config.getString("bootstrapServer
```

```
s"))
      .withGroupId("grupoAPIs")

.withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest")

.withProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
"true")

.withProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CO
NFIG, "5000")

  def iniciarStreams(): Unit = {

    val grafo = Flow.fromGraph(GraphDSL.create() {
implicit builder =>

      import GraphDSL.Implicits._

      val difusorMensagemKafka =
builder.add(Broadcast[Conta](2))
      val juntorMensagemsAPIs = builder.add(Zip[Int,
Int])

      val enviaParaFiscal = Flow[Conta].map(
        msg => {
          println(s"processando a conta
${msg.toJson.compactPrint} para o fiscal")
          (HttpRequest(method = HttpMethods.POST,
            uri =
s"http://${config.getString("url")}:3000/fiscal",
            entity =
HttpEntity(ContentTypes.`application/json`,
msg.toJson.compactPrint)), NotUsed)
        }
      )
        .via(Http().superPool[NotUsed]())
```

```
        .map(
          resposta => {
            println(s"resposta do fiscal: $resposta")
            resposta._1.get.status.intValue()
          }
        )

      val enviaParaCredito = Flow[Conta].map(
        msg => {
          println(s"processando a conta
${msg.toJson.compactPrint} para o credito")
          (HttpRequest(method = HttpMethods.POST,
            uri =
s"http://${config.getString("url")}:3000/credito",
            entity =
HttpEntity(ContentTypes.`application/json`,
msg.toJson.compactPrint)), NotUsed)
        }
      )
        .via(Http().superPool[NotUsed]())
        .map(
          resposta => {
            println(s"resposta do credito: $resposta")
            resposta._1.get.status.intValue()
          }
        )

      difusorMensagemKafka ~> enviaParaFiscal ~>
juntorMensagemsAPIs.in0
      difusorMensagemKafka ~> enviaParaCredito ~>
juntorMensagemsAPIs.in1

      FlowShape(difusorMensagemKafka.in,
juntorMensagemsAPIs.out)

    })
    Consumer
```

```
      .plainSource(
        consumerSettings,
        Subscriptions.topics("contas")
      ).map(registro => {
      val linhaComQuebraDelinha = registro.value() +
"\n"
      ByteString(linhaComQuebraDelinha)
    }
    )
      .via(CsvParsing.lineScanner())

.via(CsvToMap.withHeadersAsStrings(StandardCharsets.UTF
_8, "nome", "idade", "documento"))
      .map(registro => Conta(registro("nome"),
registro("idade").toInt, registro("documento").toLong))
      .map(conta => {
        println(s"processando a conta $conta")
        conta
      })
      .via(grafo)
      .map(
        respostas =>
          if (respostas._1 != 200 || respostas._2 !=
200) {
            println("Problema para acessar o fiscal ou
o credito! Favor checar os logs.")
          } else {
            println("Conta processada com sucesso!")
          }
      )

.withAttributes(ActorAttributes.supervisionStrategy(dec
ider))
      .runWith(Sink.ignore)
  }

}
```

E este é o nosso código refatorado:

```
package com.casadocodigo.streams

import akka.NotUsed
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.kafka.scaladsl.Consumer
import akka.kafka.{ConsumerSettings, Subscriptions}
import akka.stream.alpakka.csv.scaladsl.{CsvParsing,
CsvToMap}
import akka.stream.scaladsl.{Broadcast, Flow, GraphDSL,
Sink, Source, Zip}
import akka.stream.{ActorAttributes, FlowShape,
Supervision}
import akka.util.ByteString
import com.casadocodigo.Boot.{config, system}
import com.casadocodigo.streams.KafkaParaBanco.Conta
import com.typesafe.config.Config
import org.apache.kafka.clients.consumer.
{ConsumerConfig, ConsumerRecord}
import
org.apache.kafka.common.serialization.StringDeserialize
r
import spray.json._

import java.nio.charset.StandardCharsets
import scala.concurrent.Future
import scala.language.postfixOps
import scala.util.Try

object KafkaParaAPIs extends SerializadorJSON {

  private def chamarHttp(request: HttpRequest):
Future[Seq[Try[HttpResponse]]] = {
    Source.single((request, NotUsed))
      .via(Http().superPool[NotUsed]())
```

```
      .map(_._1)
      .runWith(Sink.takeLast(1))
  }


  private val decider: Supervision.Decider = _ =>
Supervision.Restart

  private def grafo(executorChamadaHttp: HttpRequest =>
Future[Seq[Try[HttpResponse]]])(implicit config:
Config):
  Flow[Conta, (Int, Int), NotUsed] =
Flow.fromGraph(GraphDSL.create() { implicit builder =>

    import GraphDSL.Implicits._

    val difusorMensagemKafka =
builder.add(Broadcast[Conta](2))
    val juntorMensagemsAPIs = builder.add(Zip[Int,
Int])

    val enviaParaFiscal = Flow[Conta].mapAsync(1)(
      msg => {
        println(s"processando a conta
${msg.toJson.compactPrint} para o fiscal")
        executorChamadaHttp(HttpRequest(method =
HttpMethods.POST,
          uri =
s"http://${config.getString("url")}:3000/fiscal",
          entity =
HttpEntity(ContentTypes.`application/json`,
msg.toJson.compactPrint)))

      }
    ).map(
      resposta => {
        println(s"resposta do fiscal: $resposta")
```

```
        resposta.head.get.status.intValue()
      }
    )

    val enviaParaCredito = Flow[Conta].mapAsync(1)(
      msg => {
        println(s"processando a conta
${msg.toJson.compactPrint} para o credito")
        executorChamadaHttp(HttpRequest(method =
HttpMethods.POST,
          uri =
s"http://${config.getString("url")}:3000/credito",
          entity =
HttpEntity(ContentTypes.`application/json`,
msg.toJson.compactPrint)))
      }
    ).map(
      resposta => {
        println(s"resposta do credito: $resposta")
        resposta.head.get.status.intValue()
      }
    )

    difusorMensagemKafka ~> enviaParaFiscal ~>
juntorMensagemsAPIs.in0
    difusorMensagemKafka ~> enviaParaCredito ~>
juntorMensagemsAPIs.in1

    FlowShape(difusorMensagemKafka.in,
juntorMensagemsAPIs.out)

  })

  def fluxoDeTransformacaoDados(executorChamadaHttp:
HttpRequest => Future[Seq[Try[HttpResponse]]] =
chamarHttp)
                               (implicit config:
```

```
Config): Flow[ConsumerRecord[String, String], Unit,
NotUsed] = {
    Flow[ConsumerRecord[String, String]].map(registro
=> {
      val linhaComQuebraDelinha = registro.value() +
"\n"
      ByteString(linhaComQuebraDelinha)
    }
    )
      .via(CsvParsing.lineScanner())

.via(CsvToMap.withHeadersAsStrings(StandardCharsets.UTF
_8, "nome", "idade", "documento"))
      .map(registro => Conta(registro("nome"),
registro("idade").toInt, registro("documento").toLong))
      .map(conta => {
        println(s"processando a conta $conta")
        conta
      })
      .via(grafo(executorChamadaHttp))
      .map(
        respostas =>
          if (respostas._1 != 200 || respostas._2 !=
200) {
            println("Problema para acessar o fiscal ou
o credito! Favor checar os logs.")
          } else {
            println("Conta processada com sucesso!")
          }
      )

.withAttributes(ActorAttributes.supervisionStrategy(dec
ider))
  }

  def iniciarStreams(): Unit = {
    val consumerSettings =
```

```
      ConsumerSettings(system, new StringDeserializer,
new StringDeserializer)

.withBootstrapServers(config.getString("bootstrapServer
s"))
        .withGroupId("grupoAPIs")

.withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest")

.withProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
"true")

.withProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CO
NFIG, "5000")

    Consumer
      .plainSource(
        consumerSettings,
        Subscriptions.topics("contas")
      )
      .via(fluxoDeTransformacaoDados())
      .runWith(Sink.ignore)
  }

}
```

Agora, vamos criar o nosso teste, através da classe `KafkaParaAPIsSpec`. Nesta classe, criamos um mock para a chamada das APIs HTTP, que será chamado no lugar da implementação `chamarHttp` que criamos anteriormente.

Para isso, utilizamos a biblioteca `scalamock` e criamos um mock de função através do objeto

`mockFunction`. Nesse objeto, passamos como parâmetros de tipagem qual é o tipo da nossa entrada, no caso um objeto do tipo `HttpRequest`, e qual o tipo da nossa saída, no caso um objeto do tipo `Future[Seq[Try[HttpResponse]]]`.

O `scalamock` suporta dois modelos de configuração de mocks, chamados `Expectations-First Style` e `Record-then-Verify`. No modelo `Record-then-Verify`, configuramos da maneira tradicional de outras bibliotecas, como o `Mockito`, por exemplo, onde primeiro configuramos os mocks e depois configuramos verificações de como ou se nossos mocks foram invocados.

Já no modelo `Expectations-First Style`, a configuração dos mocks já é feita junto com as verificações, neste caso melhor dizendo, *expectativas*. Estamos dizendo, em formato de código, por exemplo: *minha expectativa é de que meu mock seja chamado com qualquer valor de entrada, retorne o valor X de saída e seja invocado quatro vezes*.

É o que fazemos no trecho de código a seguir:

```
mock expects * returning Future(Seq(Try[HttpResponse] {
      HttpResponse(status = StatusCodes.OK, entity = "
{\n  \"success\": true\n}")
    })) repeat 4
```

Estamos configurando que nosso mock espera ser chamado quatro vezes, recebendo qualquer coisa como entrada - através do caractere "*" -, retornando uma `Future[Seq[Try[HttpResponse]]]` que fornecemos na configuração da expectativa.

Assim, temos o código completo da nossa classe de testes, onde criamos o mock, definimos uma expectativa, definimos `TestSource` e `TestSink` para a nossa stream e a testamos, enviando mensagens e esperando pelos resultados. Neste caso, ao final da nossa stream, não temos dados finais - a stream apenas loga informações no console -, por isso mapeamos o `TestSink` com o tipo `None` e utilizamos os caracteres `()` para representar o retorno do tipo `Unit`. O conteúdo a seguir compõe todo o código de testes que acabamos de estudar:

```
package com.casadocodigo.streams

import akka.actor.ActorSystem
import akka.http.scaladsl.model.{HttpRequest,
HttpResponse, StatusCodes}
import akka.stream.scaladsl.Keep
import akka.stream.testkit.scaladsl.{TestSink,
TestSource}
import
com.casadocodigo.streams.KafkaParaAPIs.fluxoDeTransform
acaoDados
import com.typesafe.config.{Config, ConfigFactory}
import org.apache.kafka.clients.consumer.ConsumerRecord
```

```
import org.scalamock.scalatest.proxy.MockFactory
import org.scalatest.BeforeAndAfterAll
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should

import scala.concurrent.{ExecutionContextExecutor,
Future}
import scala.language.postfixOps
import scala.util.Try

class KafkaParaAPIsSpec extends AnyFlatSpec with
BeforeAndAfterAll with should.Matchers with MockFactory
{

  implicit val system: ActorSystem =
ActorSystem("AkkaStreams")
  implicit val ec: ExecutionContextExecutor =
system.dispatcher
  implicit val config: Config =
ConfigFactory.load(Option(
    System.getenv("ENVIRONMENT"))

.getOrElse(Option(System.getProperty("ENVIRONMENT"))
      .getOrElse("application")))

  override def afterAll(): Unit = system.terminate()

  "A stream do kafka para APIs" should "ler dados do
kafka para envio para as APIs" in {

    val mock = mockFunction[HttpRequest,
Future[Seq[Try[HttpResponse]]]]

    mock expects * returning
Future(Seq(Try[HttpResponse] {
      HttpResponse(status = StatusCodes.OK, entity = "
{\n  \"success\": true\n}")
```

```
    })) repeat 4

    val (entrada, saida) =
TestSource.probe[ConsumerRecord[String, String]]
      .via(fluxoDeTransformacaoDados(mock))
      .toMat(TestSink[Unit]())(Keep.both).run()


    saida.request(2)
    entrada.sendNext(new ConsumerRecord[String, String]
("contas", 0,
      0, "chave", "\"Alexandre\",39,112322211"))
    entrada.sendNext(new ConsumerRecord[String, String]
("contas", 0,
      0, "chave", "\"Ana Carolina\",\"18\",98274391"))
    saida.expectNextUnordered((), ())

  }

}
```

Por fim, ao executar o teste, podemos ver que ele ocorre sem problemas, percorre toda a stream, efetua o processamento e chama o nosso mock na quantidade de vezes esperada:

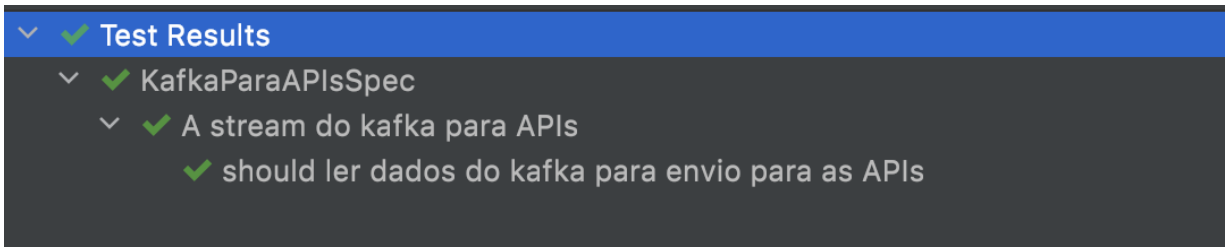



Figura 9.10: Resultado da execução dos testes da stream de Kafka para APIs.

Mas será mesmo que as nossas expectativas estão funcionando corretamente? Vamos fazer um pequeno teste mudando a quantidade de vezes que esperamos que o nosso mock seja invocado para um valor incorreto, por exemplo, 6:

```
mock expects * returning Future[HttpResponse] {
      HttpResponse(status = StatusCodes.OK, entity = "
{\n  \"success\": true\n}")
    } repeat 6
```

Ao executarmos novamente o teste, podemos ver que o teste falha e, no terminal, teremos uma exceção indicando a quantidade de vezes que esperávamos que o nosso mock fosse invocado e a quantidade de vezes que ele foi de fato invocado:

```
Unsatisfied expectation:

Expected:
inAnyOrder {
  MockFunction1-1(*) 6 times (called 4 times -
UNSATISFIED)
}
```

Assim, aprendemos como utilizar os kits de desenvolvimento de testes tanto do Akka quanto do Akka Streams, bem como alguns componentes de populares bibliotecas de testes do Scala, `scalatest e scalamock`.

Foi um longo caminho até aqui. Antes de encerrarmos, vamos refletir algumas considerações finais, bem como próximos passos.

## Capítulo 10
# Conclusão

Percorremos um longo caminho até aqui, mas foi muito gratificante. Aprendemos sobre diversas bibliotecas e conceitos, além de novos modelos de programação (modelo de atores e streams).

Paralelismo é um recurso muito poderoso e cada vez mais importante no mundo de hoje. Com as barreiras de hardware cada vez mais vencidas - através de recursos como virtualização e gerenciamento de máquinas virtuais, como o *Kubernetes* - trabalhamos cada vez mais em um mundo distribuído, onde aplicações executam como vários nós independentes processando dados em simultâneo.

Nesse mundo, frameworks como Akka permitem que exploremos paralelismo dentro dos nós das aplicações também, além do paralelismo natural fornecido por infraestruturas que utilizam tecnologias como o já citado Kubernetes. Tal recurso é muito útil para maximizar a utilização dos nossos recursos, bem como maximizar a escalabilidade dos nossos sistemas. A imagem a seguir ilustra esse ambiente em execução na prática, com requisições percorrendo os diferentes nós e dentro dos nós:

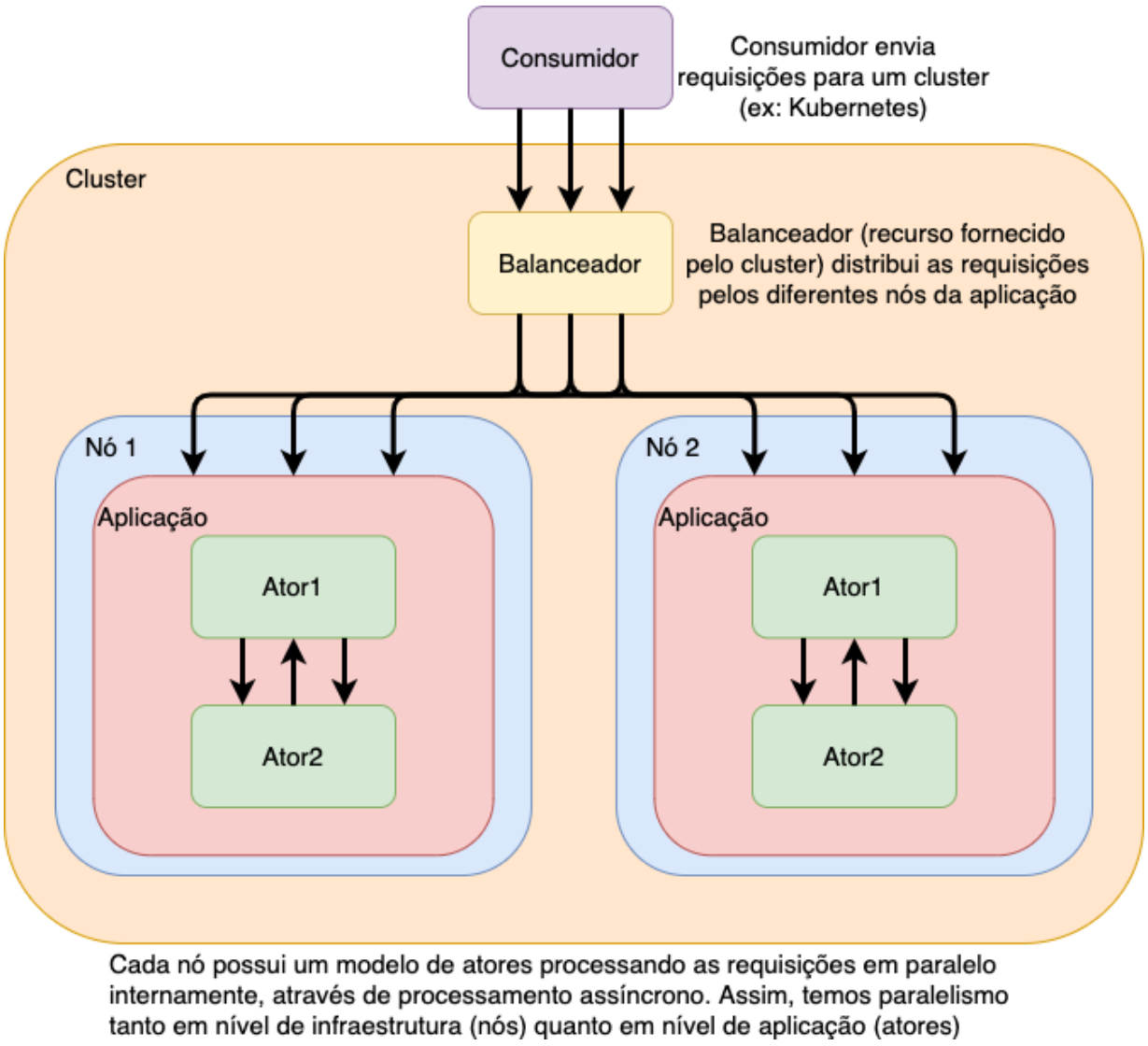

Consumidor
Consumidor envia requisições para um cluster (ex: Kubernetes)
Cluster
Balanceador
Balanceador (recurso fornecido pelo cluster) distribui as requisições pelos diferentes nós da aplicação
Nó 1
Aplicação
Ator1
Ator2
Nó 2
Aplicação
Ator1
Ator2
Cada nó possui um modelo de atores processando as requisições em paralelo internamente, através de processamento assíncrono. Assim, temos paralelismo tanto em nível de infraestrutura (nós) quanto em nível de aplicação (atores)

Figura 10.1: Execução de aplicações Akka dentro de um cluster.

Além dos benefícios de escalabilidade, também observamos como nos beneficiamos com o incremento na robustez das nossas aplicações. Vimos no capítulo 1 a pobre história dos desenvolvedores utilizando uma arquitetura tradicional não distribuída em suas aplicações, que

fez com que o sistema entrasse em colapso. Entretanto, no capítulo 5, vimos uma proposta de API muito mais robusta, que não colapsaria no cenário do início do livro, conforme vimos nos testes do Gatling. Tudo isso graças aos mecanismos de buffers e isolamento dos atores que fazem com que as partes mais custosas de uma aplicação possam ser isoladas, limitando o seu impacto na aplicação como um todo.

Antes de finalizarmos, vamos ver brevemente alguns tópicos que são bons pontos complementares para expandirmos adiante nossos estudos nessas tecnologias.

O Akka possui uma biblioteca que permite criarmos clusters de atores Akka. Nesses clusters, temos modelos de atores que se comunicam entre si de forma remota através do protocolo tcp , permitindo que façamos com que o processamento utilize atores "locais", que executem dentro da própria aplicação, e também atores remotos, localizados em outras instâncias da aplicação no cluster.

Em minhas experiências pessoais, confesso que não senti grande necessidade de usar essa biblioteca, dados os recursos que já recebemos clusterizando o mesmo modelo de atores em diferentes nós de um cluster Kubernetes, por exemplo. De todo modo, é

importante conhecermos tudo o que temos disponível para os nós de clusters em nossas ferramentas, por isso vale o estudo, ainda que por curiosidade. Mais informações podem ser encontradas em: https://doc.akka.io/docs/akka/2.5.32/index-cluster.html

Um recurso interessante que acabamos não vendo nos exemplos anteriores do nosso estudo, mas do qual cabe uma rápida demonstração, é a *personificação* (become). Com ela, conseguimos trocar a implementação de um ator em tempo de execução, fornecendo uma nova implementação. Por exemplo, se modificássemos o nosso ator que simulava chamadas a APIs do capítulo 3 da seguinte forma:

```
package com.casadocodigo.actors

import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors
import com.casadocodigo.actors.DBActor.MensagemBanco

object APIActor {

  case class MensagemAPI(nome: String, documento:
String)

  def apply(): Behavior[MensagemAPI] = Behaviors.setup
{ contexto =>
```

```
    val refAtorDB = contexto.spawn(DBActor(),
"DBActor")

    Behaviors.receive {
      (contexto, mensagem) =>
        contexto.log.info(s"mensagem $mensagem recebida
e enviada para a API!")
        refAtorDB ! MensagemBanco(nome = mensagem.nome,
documento = mensagem.documento)
        Behaviors.receive {
          (contexto, mensagem) =>
          contexto.log.info(s"mensagem $mensagem
recebida e processada sem envios externos!")
          Behaviors.same
        }
    }
  }

}
```

Poderemos ver ao executar que, a partir da segunda mensagem enviada para o ator, a mensagem de log é modificada, demonstrando a troca da implementação:

```
20:47:04.260 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO com.casadocodigo.actors.APIActor$ -
mensagem MensagemAPI(teste,123456) recebida e enviada
para a API!
20:47:04.262 [MyActorSystem-akka.actor.default-
dispatcher-6] INFO com.casadocodigo.actors.DBActor$ -
mensagem MensagemBanco(teste,123456) recebida e
persistida no banco!
20:47:04.262 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO com.casadocodigo.actors.APIActor$ -
mensagem MensagemAPI(teste,123456) recebida e
```

```
processada sem envios externos!
20:47:04.264 [MyActorSystem-akka.actor.default-
dispatcher-7] INFO com.casadocodigo.actors.APIActor$ -
mensagem MensagemAPI(teste,123456) recebida e
processada sem envios externos!
```

Outro recurso interessante de se estudar é o *circuit breaker*. Circuit breaker é um *design pattern* (padrão de projeto) onde definimos uma rota de falha para casos em que a chamada a um determinado recurso começa a falhar consistentemente. Nesse cenário, dizemos que o circuito foi **aberto**.

Após a abertura do circuito, a rota de falha é utilizada em todas as requisições, eventualmente realizando uma nova chamada para o recurso a fim de verificar se ele voltou a funcionar. Em caso de nova falha, o circuito é mantido aberto, caso contrário, o circuito muda para o estado que chamamos de **fechado**, em que as requisições param de usar a rota de falha e voltam a utilizar o recurso.

A imagem a seguir ilustra o funcionamento do padrão de projeto. Para o Akka, detalhes sobre a sua implementação podem ser encontrados em:

https://doc.akka.io/docs/akka/2.5.32/common/circuitbreaker.html

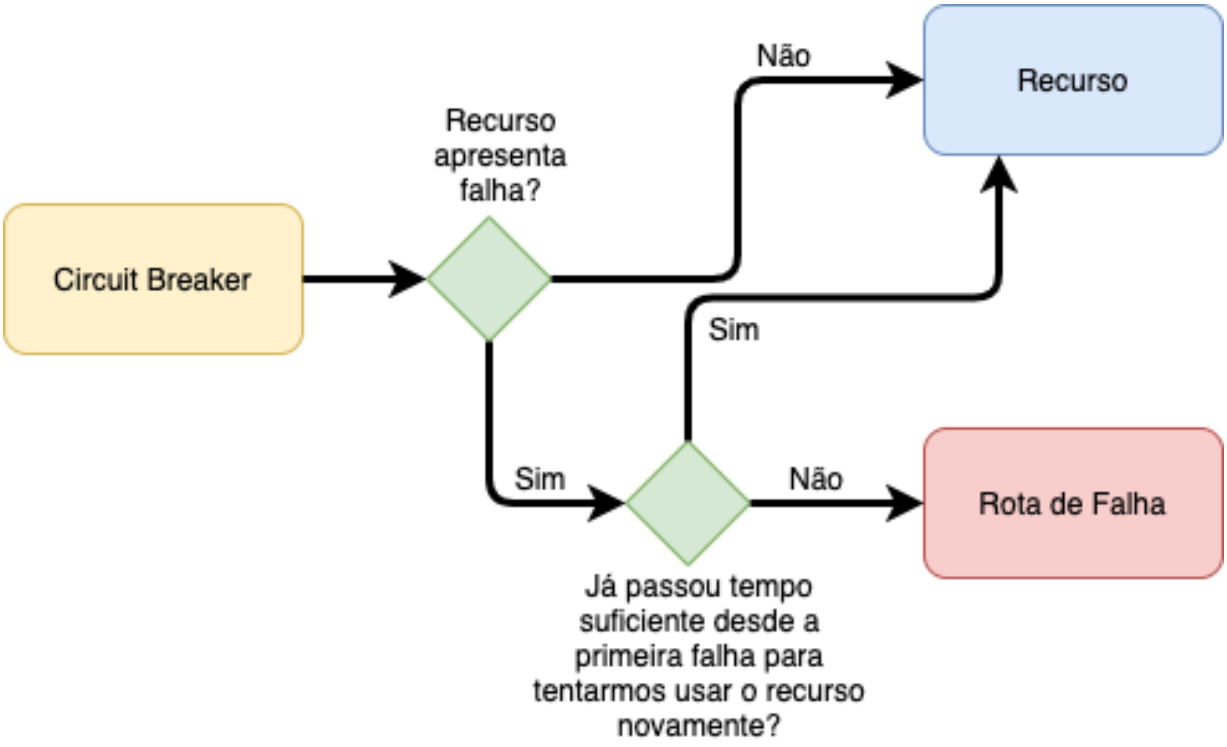



Figura 10.2: Máquina de estados do circuit breaker.

Outro assunto interessante a se explorar são os tipos de canais (caixas de correios) do Akka. Para a maioria esmagadora dos cenários, os canais padrão do Akka já atendem muito bem a todos os cenários. Esses canais consistem de canais do tipo *unbounded* (sem limite), ou seja, enquanto mensagens são recebidas e o consumo ainda não conseguiu processar todas as mensagens do canal, as mensagens são empilhadas infinitamente.

O Akka nos fornece outros tipos que podem ser interessantes em situações bastante específicas de aplicações, como um canal que implementa uma fila

de prioridade. Mais informações podem ser encontradas em: https://doc.akka.io/docs/akka/current/typed/mailboxes.html#mailbox-implementations

Finalizando o assunto sobre a expansão do nosso estudo, um outro ponto interessante é o teste de rotas do Akka HTTP. Esse estilo de teste, considerado um teste de integração, é um complemento para as bibliotecas principais de testes que vimos anteriormente e é muito simples de se utilizar. Mais informações sobre ele podem ser encontradas em: https://doc.akka.io/docs/akka-http/current/routing-dsl/testkit.html

E assim chegamos ao fim do nosso aprendizado. Agradeço muito a você por ter lido este livro e espero que o conhecimento compartilhado aqui possa ser aplicado em seu trabalho ou estudos. Até a próxima!